Machine Learning Project
Final Report

Arthur Ballegeer, Nicolas De Bie, Maarten Vanmarcke

May 17, 2023

1 Introduction

Many applications involve multiple agents. It is important for artificially intelligent agents to take the presence of these other agents into account. This paper explores the application of multi-agent reinforcement learning (MARL) in various scenarios. Our study begins in Section 2 by examining four benchmark matrix games. We employ three distinct algorithms to analyze the evolution of multiple agents' strategies. The learning trajectories are visualised for all algorithms and analyzed with a particular focus on assessing their correspondence to replicator dynamics. Furthermore, Nash equilibria and Pareto optimality points, originating from Game Theory, are used to provide insights into the observed plots. Moving forward in Section 3, we delve into solving the smallest version of the Dots-and-Boxes game using four different algorithms. Detailed comparisons are made among execution times in relation to the game state space sizes and the characteristics of the algorithms. Lastly, this study tackles the implementation of an agent using deep reinforcement learning for the Dots-and-Boxes game in Section 4. The goal is to create an agent that is capable of playing both a relatively large board size (e.g. 7x7) as well as an undisclosed game size, emphasizing the importance of adaptability to varying board dimensions. An extra constraint is that the agent must be able to make a move in a reasonable amount of time, i.e. about 100 msec per move. Our study ends in Section 5 by summarizing all the conclusions for each task.
For each task we begin by summarizing the academic work that helped and inspired our implementations. After this, we go over our methodology and design decisions. We end each section by showing, examining and discussing the obtained results. To implement everything, we made use of the OpenSpiel open-source software framework.[1]

2 Task 2 - Learning & Dynamics

Matrix games are simple, benchmark games. They can be used to examine the behaviour of learners. We want to find out to which pair of strategies the learners converge for different kind of matrix games and how these learning trajectories differ for three different algorithms: ϵ-greedy, Lenient Boltzmann Q-learning and cross learning.

2.1 Literature Study

As the literature study for Task 2 is quite broad, we concisely run through the various topics seperately:

- **Q-Learning** – Q-learning is an example of a reinforcement learning algorithm.[1, 2] The agent keeps track of a Q-value for each state-action pair. If the agent is in state s_t and performs action a_t, then it will update that Q-value $Q(s_t, a_t)$. The action to take (the policy) is determined by taking the action that yields the highest Q-value for the given state s, i.e. $\arg\max_a Q(s, a)$. It can be theoretically proven that this converges to the optimal policy, given sufficient updates for each pair and a decreasing learning rate.

- **Exploration** – While training an agent, different strategies exist for choosing the next action. With exploitation, the agent selects the current best action (highest Q-value) to try to maximize its reward. Exploration means discovering new actions and their rewards to improve the policy. A good balance is needed to become an experienced agent that has no weak spots.[2] Different methods exist to get a good balance between the two.[3] One of the two methods that we use, is ϵ-greedy exploration, where the agent selects a random action with a probability of ϵ, otherwise the current best action is chosen (highest Q-value).[2] The other exploration

[1] `https://github.com/deepmind/open_spiel`

method that we use, is Boltzmann exploration where probabilities are assigned to each action as the softmax weights of their Q-value.[2] τ, a temperature parameter, can be increased to spread the distribution and thus increase the amount of exploration.

- **Cross learning** – Cross learning is another reinforcement learning algorithm.[1] It is often used to uncover the link between reinforcement learning and evolutionary game theory. In cross learning, each agent keeps track of a normalised probability $\pi(i)$ for each action a_i. The probability that the agent executes action a_i is then given by that probability $\pi(i)$ and is independent of the given state. If action a_i is executed and gives a good reward, then $\pi(i)$ gets increased and all $\pi(j)$ for all $j \neq i$ decreased and vice versa.

- **Leniency** – It is possible that agents get stuck in a suboptimal policy when learning. Leniency provides a solution by only updating the utilities of the actions that resulted in a higher reward and ignoring those that resulted in lower rewards.[4] A possible way to achieve this is by updating the utility only if it is lower than the observed reward. Another possible way is by observing κ rewards for a certain action and the maximum reward of these κ rewards is used to update the utility for this action. Thereafter, those κ rewards are flushed and forgotten.

- **Nash Equilibrium** – In 1951 John Nash introduced a new solution concept for non-cooperative games, i.e. games where players make their own decisions independently without any communication.[5] In this paper, Nash proposes a Nash equilibrium as a strategy profile where no player can unilaterally improve their strategy, given that all other players are following their own strategies. In other words, each player's strategy has the highest expected pay-off if other players do not change their strategies. We can think of this equilibrium as a non-cooperative solution of the game.[6]

- **Pareto Optimality** – A Pareto optimal solution is a pair (if there are two players) of strategies for which no player can improve its pay-off without decreasing the pay-off of the other player. In other words, in a point that is not a Pareto optimal solution, a player can increase its pay-off without harming the other players. Therefore, the players are considered to cooperate.[6]

- **Replicator Dynamics** – Replicator dynamics describe how a population evolves over time, by means of interactions between individuals of these populations.[1] This evolution is characterized by

$$\dot{x_i} = x_i \left[f_i(\mathbf{x}) - \bar{f}(\mathbf{x}) \right]$$

where $\mathbf{x}$ is the distribution of the population groups, $f_i(\mathbf{x})$ is a fitness function for group i and $\bar{f}(\mathbf{x})$ is the average fitness of the population. For matrix games, the fitness is given by the pay-off matrices. This results in the following equations:

$$\dot{x_i} = x_i \left[(\mathbf{Ay})_i - \mathbf{x}^{\mathrm{T}}\mathbf{Ay} \right] \qquad \dot{y_i} = y_i \left[(\mathbf{x}^{\mathrm{T}}\mathbf{B})_i - \mathbf{x}^{\mathrm{T}}\mathbf{By} \right]$$

with $\mathbf{x}$ for the probabilities of the actions of player 1 and $\mathbf{y}$ for those of player 2 and with $\mathbf{A}$ and $\mathbf{B}$ for the pay-off matrices of player 1 respectively player 2. It can be proven that the cross learning algorithm follows these dynamics for infinitely small step sizes.

2.2 Methodology

For the different algorithms, we used the available implementations in the OpenSpiel library[2].
In Table 1, we list the parameter values we used to make the plots in Section 2.3. The number of iterations was set through trial-and-error to guarantee an optimal visualisation of the convergence or divergence. The criterium to choose the step size was to obtain a trajectory that is sufficiently fast while maintaining a smooth curve. A larger step size increases the speed of convergence but results in a rough trajectory. The trade-off in tuning κ for Lenient Boltzmann is between exploration (which can be observed by little deviations in the trajectory) and fast learning. We varied the parameter κ, but this did not have a significant influence on the figures. The parameters ϵ and τ are parameters that control the balance between exploitation and exploration as discussed in Section 2.1. In the beginning, exploration needs to be ensured. However after a while the exploitation needs to be dominating and a decay of exploration is advised.[7] Therefore we chose a linear schedule for these parameters.

[2] `https://github.com/deepmind/open_spiel`

Algorithm	Parameter	Value
ϵ-greedy	ϵ	0.3-0.05 (linear schedule)
	Step size	0.01
	Iterations	1000
Lenient Boltzmann	τ	0.3-0.01 (linear schedule)
	κ	10
	Step size	0.001
	Iterations	100 000
Cross	Step size	0.0005
	Iterations	200 000

Table 1: The parameter values used for generating the plots in Section 2.3.

Symbol	Red dot	Yellow diamond	Green dot
Meaning	Starting point	Nash equilibrium	Pareto optimal solution

Table 2: General legend for the learning trajectories for all algorithms and all games in the plots in Section 2.3.

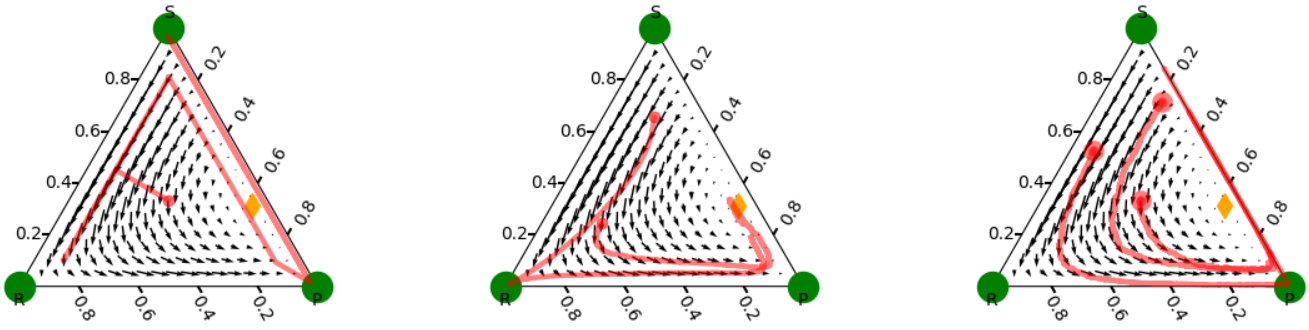
2.3 Results and Discussion

2.3.1 Biased Rock-Paper-Scissors

The biased Rock-Paper-Scissors game has one (mixed) Nash equilibrium, namely in R = $\frac{1}{16}$, P = $\frac{10}{16}$ and S = $\frac{5}{16}$ for both players. This point is also a Pareto optimal solution. All pure strategies are also Pareto optimal solutions. When looking at the replicator dynamics for the biased Rock-Paper-Scissors game in Figure 1, we observe that all three learning algorithms cycle around the Nash equilibrium, but only the Lenient Boltzmann Q-learner converges in it. We expect that this cycling is due to the influence of the three Pareto optimal solutions around the Nash equilibrium.

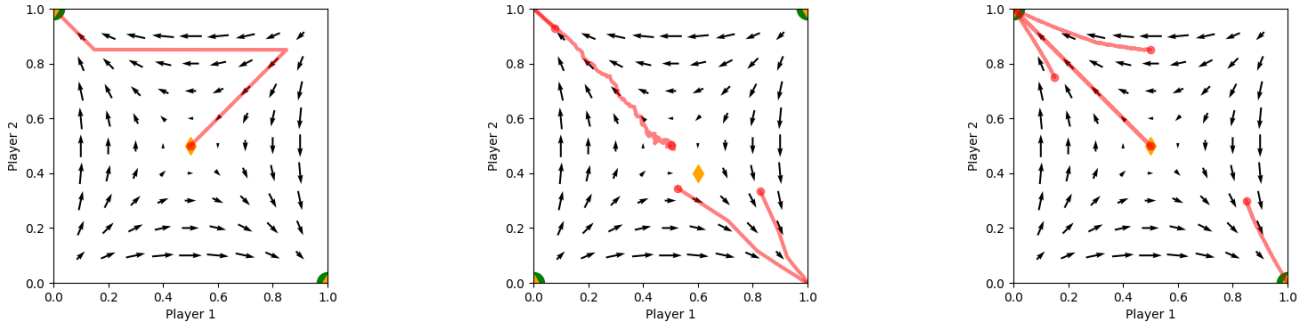
2.3.2 Dispersion Game

We can see in Figure 2 that for all algorithms the learning trajectories evolve to one of the stable, pure Nash equilibria (0,1) or (1,0), depending on their starting positions. These are also Pareto optimal points. This is because the Dispersion Game is a coordination game where the mutual and collective rewards are optimal (and suboptimal) for the same outcomes. The trajectories never converge to the unstable, mixed Nash equilibrium (0.5,0.5). This equilibrium is not a Pareto optimum. We think that the learning trajectories want to converge to a Nash equilibrium, but they are attracted by the Pareto optimal solutions. That is why they do not converge to a Nash equilibrium that is not a Pareto optimal solution.



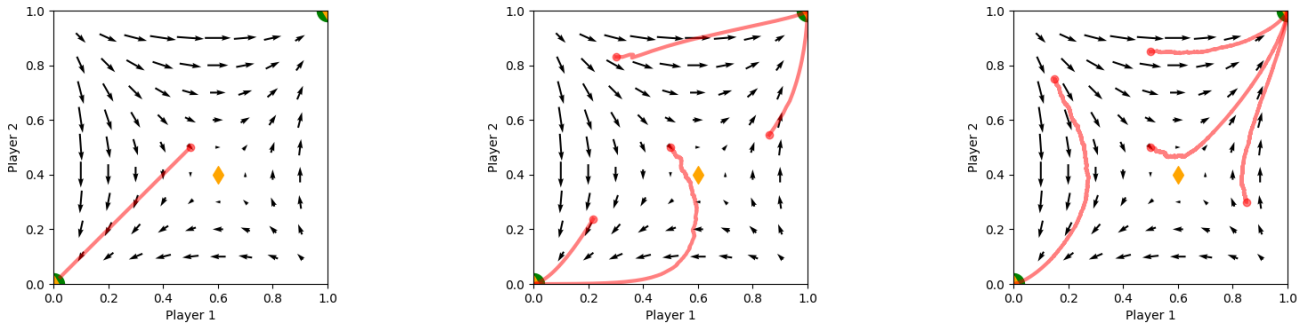
(a) (b) (c)

Figure 1: (a) Epsilon greedy Q-learning (b) Lenient Boltzmann Q-learning (c) Cross learning. Learning trajectories (color) and field plots (black) for the Biased Rock-Paper-Scissors. A legend can be found in Table 2.



(a) (b) (c)

Figure 2: (a) Epsilon greedy Q-learning (b) Lenient Boltzmann Q-learning (c) Cross learning. Learning trajectories (color) and field plots (black) for the Dispersion Game. A legend can be found in Table 2.



(a) (b) (c)

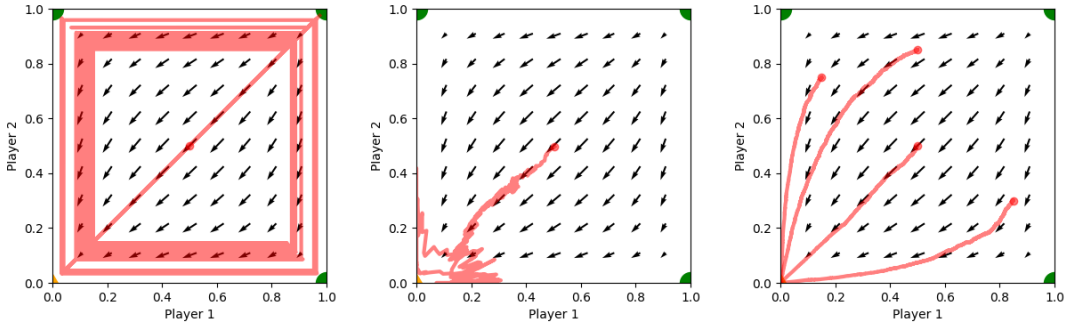
Figure 3: (a) Epsilon greedy Q-learning (b) Lenient Boltzmann Q-learning (c) Cross learning. Learning trajectories (color) and field plots (black) for the Battle of the Sexes. A legend can be found in Table 2.

2.3.3 Battle of the Sexes

For the Battle of the Sexes game, all algorithms lead to one of the stable, pure Nash equilibria in (0,0) and (1,1), as can be seen in Figure 3. Both these points are Pareto optimal. This is another example of a coordination game, therefore the pure Nash equilibria are Pareto optimality points (strategies). Again, the algorithms do not converge to the unstable, mixed Nash equilibruim in (0.6, 0.4). This is not a Pareto optimality point. We observe again that the trajectories try to converge to a Nash equilibrium and are attracted by Pareto optimums.

2.3.4 Prisoner's Dilemma

Once again all the learning trajectories converge towards the (only) pure Nash equilibrium for the Prisoner's Dilemma in (0,0), meaning both players choose to defect (i.e. confess, betray partner), shown in Figure 4. This is because for both players in both cases, the optimal pay-off is to defect. This time, this is not the Pareto optimality point. The Pareto optimality points are (1,1) (both players stay silent), (0,1) and (1,0). This is typical for social dilemma games. In Figure 4b we can see that the learning trajectory takes some sidesteps before fully converging towards the Nash equilibrium. This could be because of this dilemma between the Pareto optimality point and the Nash equilibrium. The lenient part of the learning algorithm pushes the agent to take sidesteps (exploration) from its best (current) policy. This gives us again reason to believe that the trajectories try to converge to a Nash equilibrium and are attracted by Pareto optimums.



(a) (b) (c)

Figure 4: (a) Epsilon greedy Q-learning (b) Lenient Boltzmann Q-learning (c) Cross learning. Learning trajectories (color) and field plots (black) for the Prisoner's Dilemma. A legend can be found in Table 2.

3 Task 3 - Minimax for small Dots-and-Boxes

The Dots-and-Boxes game has relatively simple rules, but has a high amount of possible game states ($|S| = 2^{r(c+1)+(r+1)c}$ for a board with $c \times r$ cells). Smart algorithms are therefore needed to solve the game, even for small board sizes. We implement 3 variations of the minimax algorithm: alpha-beta pruning, minimax using a transposition table and minimax using a transposition table that exploits symmetries. We look to implement these algorithms most efficiently for the Dots-and-Boxes game and compare how they perform when trying to solve small board sizes of the Dots-and-Boxes game.

3.1 Literature Study

To solve the Dots-and-Boxes game in reasonable time for board sizes bigger than 1x1 or 1x2 better variants of the minimax algorithm are needed. A first optimization is to use transposition tables. As the Dots-and-Boxes game is impartial, saving only the current state and score suffices to solve the game without having to store which player chose which edge.[8] This makes the transposition table highly space efficient compared to other games.
An even better alternative algorithm exploits the symmetries of the game board. For the Dots-and-Boxes game most states have at most 4 symmetries: along with the original board (1), mirroring the board vertically (2), horizontally (3) and rotating the original board 180 degrees (4). If the game board is square however, another four variants are possible: rotating the orignal board 90 degrees (5), 270 degrees (6), rotating the vertically mirrored board 90 degrees right (7) and rotating the vertically mirrored board 270 degrees right (8). This could lower the amount of entries in the transposition table by 8 in case of squared boards instead of 4 for rectangular board sizes.[8]

3.2 Methodology

We solved the smallest sizes of the Dots-and-Boxes game using four algorithms. We used available implementations for the standard minimax[3] and minimax using alpha-beta pruning[4].
For both algorithms using the transposition table, we used the string representation of the state (given by the command `state.dbn_string()` in OpenSpiel) because this representation is the most space-efficient way to store all the needed information to represent the state, as mentioned in 3.1. The values are dictionaries with the score of the player thus far as key and the value of the minimax algorithm as value.
When trying to exploit symmetries, searching the transposition table for the current state is equivalent to finding one of the equivalent symmetrical states and returning that score. Rectangular boards have at most 4 symmetrical states while square boards can have up to 8, as discussed in 3.1.

[3] `https://github.com/ML-KULeuven/ml-project-2022-2023/blob/main/minimax_template.py`
[4] `https://github.com/deepmind/open_spiel/blob/master/open_spiel/python/algorithms/minimax.py`

3.3 Results and Discussion

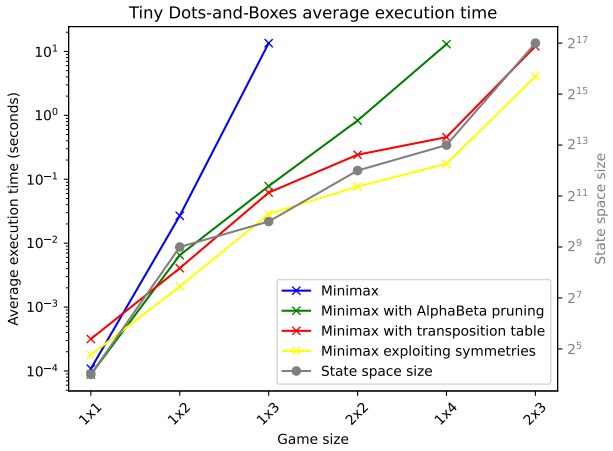
We ran all four algorithms 20 times for the 6 smallest game sizes and evaluated the average results. The solutions can be seen in Table 3.

The average execution times can be seen on Figure 5a. The execution times have been omitted if they exceeded 30 seconds. We can see that the execution times of the transposition table extensions (with or without exploiting symmetries) grow somewhat in proportion with the game state size (grey line in Figure 5a) while the execution time for the more simplistic algorithms grow faster. The cost of setting up and searching the transposition table is significant for small board sizes, but amortizes as the game board size grows. The execution time for the algorithm using transposition tables with symmetries is 1.8 times faster on average than its equivalent without symmetries for a 1x1 board and 2.60 times faster for board size 2x3. For the squared 2x2 board the symmetry-exploiting variant is even 3.15 times faster on average than the algorithm that does not exploit the symmetries, although we expected a more significant difference because of the eight total symmetrical states. This is probably because the calculation of the symmetries takes some time as well. A possible improvement could be to calculate one symmetrical state at a time and check if it is present in the transposition table. This does not require to calculate all symmetries for every move.

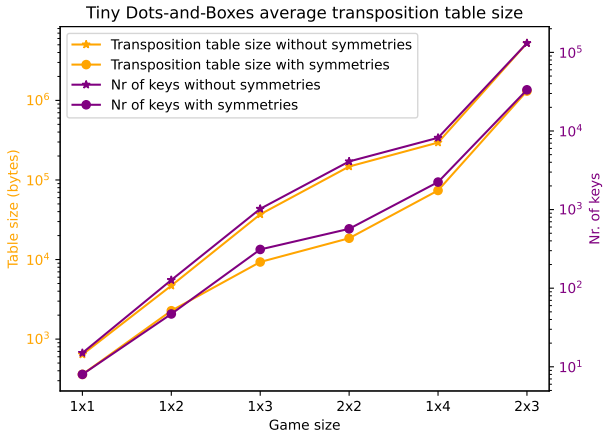
On Figure 5b we can see that the transposition table size for both algorithms (using the transposition table) is proportionate to its number of keys (the table length), with a factor relative to the size of the game state representation for each game board size. We see that the transposition table size grows exponentially (note the logarithmic axis) with the board size, which was to expect as this is proportionate to the growth of the number of states. The ratio of the transposition table length of the two algorithms approaches 4 when the game board size grows (1.87 for 1x1, 3.92 for 2x3), as we expected from the number of symmetries discussed in Section 3.1. For the squared 2x2 board size, the table length for the algorithm that does not exploit symmetries is 7.2 times bigger than for the algorithm that does. A probable explanation for this difference in theoretical value of 8 and observed value is that some states have less unique symmetrical states than 4 or 8 because one of these symmetries is equal to the original or another symmetrical state.

Game Size	1x1	1x2	1x3	2x2	1x4	2x3
Game result	Player 2 wins	Draw	Player 2 wins	Player 1 wins	Draw	Player 2 wins

Table 3: Solutions per board size for the smallest Dots-and-Boxes games if both players play optimally.



(a) Average execution time per algorithm. The state space size is given in grey.



(b) Transposition table sizes and lengths (number of keys) for the algorithms using the transposition tables.

Figure 5: Average results of solving tiny Dots-and-Boxes games 20 times for different board sizes.

4 Task 4 - Full Dots-and-Boxes

In this final task we explore a strategy to train an agent that is able to play the game of Dots-and-Boxes on larger board sizes. The same agent has to be able to play games of varying board sizes. The agent has to be fast enough to decide each move in around 100 milliseconds.

4.1 Literature Study

We decided to use MCTS to approach the problem and to use an ANN to accurately guide the MCTS and prevent it from having to perform many simulations.[9, 10] This paper [9] is useful as they take into account the runtime performance of their model. Going further down the AlphaZero path we also found an interesting open-source resource [11] explaining AlphaZero more along with open source code implementing the framework[5]. This open-source AlphaZero code was chosen as a skeleton for our agent where we later integrated our neural net and the OpenSpiel Dots-and-Boxes game representation. Lastly a paper describing a model that can generalize to multiple board sizes was found [12]. This paper proposes an AlphaZero approach that is scalable to various board sizes for the game of Go. This proposition has two main advantages for our purpose: the ability to train our model on smaller sized boards, which leads to faster training, and the ability for the model to scale to bigger board sizes.

4.2 Methodology

The general overview of our method consists of using an AlphaZero framework with a graph neural network to guide the MCTS.

4.2.1 What representation is used to represent a game state of Dots-and-Boxes?

Graph neural networks take a graph as input. Therefore the pyspiel game state needs to be converted to a graph. The paper "Solving dots-and-boxes" [8] mentions the alternative coin-and-string representation for the game and that is what we used here. The nodes represent the boxes and have a single node feature that has value 0.0 if the box is empty, 1.0 if the box has been captured by the player and -1.0 if captured by the opponent. All nodes are connected with their neighbouring nodes using an edge. The edge nodes are connected to a dummy node. This has performance benefits for the message passing algorithm during training.[12] There is an additional edge attribute for every edge. This attribute has the value 0.0 if the edge is still playable and 1.0 is the edge has been played. An example of a game state is given in Figure 6a and its corresponding graph representation is given in Figure 6b. We also added another node feature, the node valency [13].

(a) (b)

Figure 6: Dots-and-boxes game state and corresponding graph representation.

4.2.2 What model architecture is used in the agent?

The graph neural net used in our agent consists of:

- Three graph neural network layers with layer normalization and using a ReLU activation function.

[5] `https://github.com/suragnair/alpha-zero-general`

- A concatenation of all intermediate representations of these layers.
- Two fully-connected layers followed by batch normalization and a dropout layer.
- A policy-head is achieved using a fully-connected layer followed by a sigmoid function.
- A value-head is achieved using a fully-connected layer followed by a global mean pooling layer and a tanh function.

We based our architecture on the architecture mentioned in the "Train large play small" paper [12]. We tested two variations of this architecture, an architecture that takes into account both the edge features and node features, i.e. whether a node is captured, and a model that only uses node features, i.e. whether a node is captured and the node valency. Surprisingly the model only taking into account the node features outperformed the model taking into account both node and edge features. Another distinction between the two different models is the concatenation layer where the former includes the input layer in the concatenation (4 representations) whereas the latter doesn't include the input (3 representations).

4.2.3 Which training pipeline is used for the agent?

A single training iteration of our agent consisted of several episodes of self-play followed by the training of the graph net and ends with pitting the previous version against the newly trained version to determine if the new changes are accepted.

The use of graph neural networks removes the need for exploiting symmetries in the game [12]. We experimented with generating self-play examples from games played on different board sizes going from 4x4 to 7x7.

Before starting to train, we performed a grid search to decide upon the initial learning rate, batch size and number of channels regarding the neural net. The initial values can be seen in Table 4.It is advised to change the learning rate during the training process.[10, 11] In our case this was done manually by observing the loss evolution throughout the epochs. In retrospect, a learning rate scheduler should have been used. The number of channels is intuitively quite large and this does affect the time it takes for our agent to choose an action. If we had more time, the trade-off between the amount of channels and number of MCTS simulations could have been investigated.
Two other very important parameters influencing the self-play of our agent are the temperature threshold and the constant polynomial upper confidence trees (c PUCT) value. Both these parameters play a role in the balance between exploration and exploitation during self-play. Another grid search was executed to find the optimal initial values. Both the grid searches were conducted on a board size of 3x3, 5 iterations and 10 episodes per iteration.

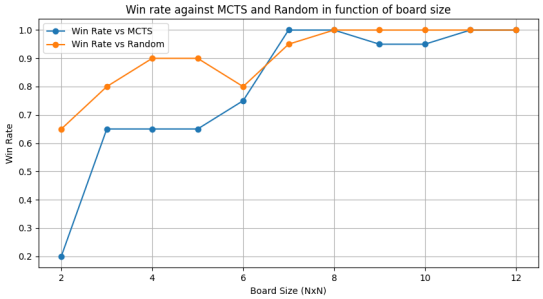
Learning rate	0.001
Batch size	64
Number of channels	512
Temperature threshold	30
c PUCT	3.0

Table 4: Initial training values.

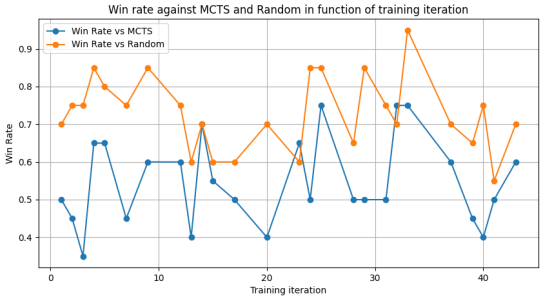
4.3 Results and Discussion

The first model we trained consisted of a GNN with 512 channels that only took the node features into account. We tried many different modifications, such as different architectures, introducing a temperature threshold, self-play on a range of board sizes,... Unfortunately and surprisingly none of these modifications ever exceeded the performance of our original model, even after numerous hours of training time.
In Figure 7a we evaluated our agent on increasing board sizes. The trouble with these evaluations is that it gets easier to win against a random bot when the board size increases. Alongside evaluating against a random bot, we also evaluated against an MCTS with random evaluator. To try and compensate for the increasing board size, the amount of MCTS simulations of this MCTS increased when increasing board size. In Figure 7a we added 10 MCTS simulations when increasing the board size with 1 row or column starting with 15 simulations for 2x2.

To evaluate our agent, we put it head to head against an agent trained using the OpenSpiel's out of the box implementation of AlphaZero with the 'conv2d' model. This model was trained for 60 iterations on a 7x7 board. When evaluating our agent (15 MCTS simulations) against the CNN model (150 MCTS simulations) our model achieves a win rate of 95% percent.



(a) Evaluation of agent on various board sizes.

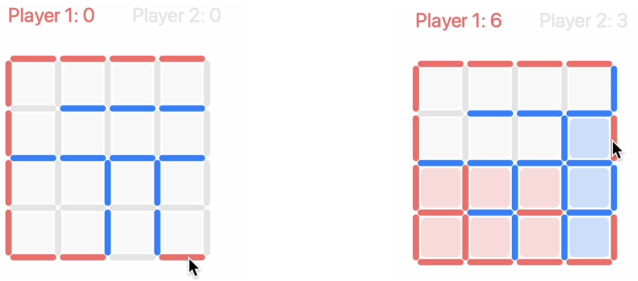


(b) Evaluation of agent during training iterations on 4x4 board size.

Figure 7: Evaluation of our Dots-and-Boxes agent.

The training process of our agent can be seen in figure 7b. This chart sums up our main difficulty: the training convergence. Every datapoint on the chart represents an iteration where, during the evaluation stage, the new version beats the older version and is thus accepted. As can be seen there are many examples where versions are accepted that perform worse against the random or MCTS evaluation benchmarks than previous versions. This is a problem we never managed to solve.

When evaluating the gameplay of our agent we can see that the agent tries to avoid giving the opposing player an opportunity to close a box for as long as possible. This can be seen in Figure 8a. Later in the game, the agent does not recognize the chain that could result in winning at least half of the available points. This can be seen in Figure 8b where the human has drawn the red edge beneath the mouse pointer and our agent closes the box and plays the edge above the mouse wasting a potential full chain. When playing against our agent, we can see that it has difficulty recognising chains, which is a crucial strategy to win the game. This makes it difficult to achieve a high win rate in the tournament.



(a) (b)

Figure 8: Examples of our agent gameplay. Our agent plays in blue.

5 Conclusion

For all the different learning algorithms in Task 2, the optimal policy tries to converge to a Nash equilibrium. We also observe that they are attracted towards the Pareto optimality points. This can be observed by the curve of their trajectory or by the specific Nash equilibria they converge to. Generally, all learning trajectories evolve towards these stable Nash equilibria quite directly, only for learners that allow some more exploration we can see some fluctuations or side steps. The difference between the games is mostly the location of the Nash equilibria.

Only for the Prisoner's Dilemma we can see a higher amount of fluctuations, certainly for the Lenient Boltzmann learner in Figure 4b. This is probably because there is no Nash equilibrium that is also a Pareto optimality point. From Task 3 we can conclude that the minimax algorithm is impossible to solve the Dots-and-Boxes game due to very long execution times for even small game board sizes (1x4, 2x2), even with alpha-beta pruning. Algorithms using transposition tables are a much better option as their execution time is proportionate to the game state size, even if the board size grows. The variant using symmetries is a factor better than the standard version, although the speed-up is less than theoretically expected.
Using graph neural nets for Task 4 was an adventurous approach and thus a gamble on the ultimate effictiveness of the outcome. We can conclude that we succesfully managed to create an agent able to play on varying board sizes. Our agent is also able to convincingly beat a number of baselines. On the other hand it is quite dissapointing that we weren't able to improve our agent to the point of it fully understanding the concept of chains.

Our code can be found on the departemental computers under student number r0810938. Maarten estimates to have worked around 120 hours on this project (50 hours on Task 2, 30 on Task 3 and 40 on Task 4). Arthur thinks he worked roughly 105 hours on the project, about 45 hours on Task 2, 50 hours on Task 3 and 10 hours on Task 4. Nicolas has worked around 110 hours on this project (20 hours on Task 2, 10 on Task 3 and 80 on Task 4).

References

[1] Daan Bloembergen, Karl Tuyls, Daniel Hennes, and Michael Kaisers. Evolutionary dynamics of multi-agent learning: A survey. *Journal of Artificial Intelligence Research*, 53:659–697, 2015.

[2] Karl Tuyls and Gerhard Weiss. Multiagent learning: Basics, challenges, and prospects. *Ai Magazine*, 33(3):41–41, 2012.

[3] Anil K Gupta, Ken G Smith, and Christina E Shalley. The interplay between exploration and exploitation. *Academy of management journal*, 49(4):693–706, 2006.

[4] Liviu Panait and Karl Tuyls. Theoretical advantages of lenient q-learners: An evolutionary game theoretic perspective. In *Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–8, 2007.

[5] JF Nash. Non-cooperative games. *Annals of mathematics*, 54(2):286–295, 1951.

[6] Richárd Kicsiny and Zoltán Varga. New algorithm for checking pareto optimality in bimatrix games. *Annals of Operations Research*, 320(1):235–259, 2023.

[7] Aakash Maroti. Rbed: Reward based epsilon decay. *arXiv preprint arXiv:1910.13701*, 2019.

[8] Joseph Barker and Richard Korf. Solving dots-and-boxes. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 26, pages 414–419, 2012.

[9] Alba Cotarelo, Vicente García-Díaz, Edward Rolando Núñez-Valdez, Cristian González García, Alberto Gómez, and Jerry Chun-Wei Lin. Improving monte carlo tree search with artificial neural networks without heuristics. *Applied Sciences*, 11(5), 2021.

[10] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[11] Surag Nair. A simple alpha(go) zero tutorial. `https://web.stanford.edu/~surag/posts/alphazero.html`, 2017.

[12] Ran El-Yaniv Shai Ben-Assayag. Train on small, play the large: Scaling up board games with alphazero and gnn. 2021.

[13] Yimeng Zhuang, Shuqin Li, Tom Vincent Peters, and Chenguang Zhang. Improving monte-carlo tree search for dots-and-boxes with a novel board representation and artificial neural networks. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 314–321, 2015.