Algorithmic Exploration of the Game Mathematico

Samuel Gazda
Masaryk University, FI
469083@mail.muni.cz

Michal Barnišin
Masaryk University, FI
485135@mail.muni.cz

Abstract

`Mathematico` is a single-player, sequential, randomised, optimisation game with a large state space and complex scoring rules. In this report[1], we introduce and analyse this game, and suggest reinforcement learning agents to solve the game. We conclude by comparing the results with the collected human performance data and proposing further improvements which might lead to super-human performance.

1 Introduction

In recent years there has been a lot of innovation in the world of the game-playing algorithms. The progress in deep learning opened many possibilities out of the reach of the classical approaches based on pure state-space exploration. One of the most successful algorithms was *AlphaZero* [2], which combines the Monte Carlo Tree Search (MCTS) and the deep neural network trained by Reinforcement learning (RL). In this work, we tried to implement a similar approach to *AlphaZero* for the game `Mathematico`.

`Mathematico` is a single-player strategy card game. The player is tasked with placing progressively drawn cards onto the board of size five by five. After the board is full, they are then rewarded with points, based on different combinations of cards placed. The rules are quite simple, but the state space of this game is large enough to make beating the human player non-trivial.

In this work, we introduce our version of *AlphaZero*-inspired algorithm and compare it to several other approaches, including the MCTS approach and human performance. In the first parts of this work, we write about the game itself and introduce several game-playing approaches. The final parts present the evaluations of the used approaches and provide the conclusion, our observations and suggestions for possible improvements.

2 Related Work

AlphaZero was created as a generalisation of *AlphaGo Zero* by the research company DeepMind. It consists of the Monte Carlo Tree Search combined with a neural network. The MCTS part searches through the game states, but instead of the typical (randomised) roll-out phase of the MCTS, this approach uses Value and Policy Heads of the network to approximate the evaluation of the analysed states and guide the MCTS to more promising states [2]. The rules of the game were incorporated into the model via the game state, rewards, and transition functions. The authors claim, that the approximation that the network provides is imperfect, but those imperfections are evened out due to the way the MCTS works with these values.

The network trained in a self-play loop. The model played the game against another instance of itself and then the network was trained on the data collected from these games. The trained *AlphaZero* model achieved state-of-the-art performance within 24 hours of training in Chess, Go and Shogi [2].

[1] This report describes the project we solved from *FI:PA026* during Spring 2023 semester.

3 Mathematico

`Mathematico` [1] is a card game, in which the player's objective is to get a high score by placing the sequentially drawn cards on the five-by-five grid. Each card has a numerical value, from one to thirteen. The deck consists of four copies of each card. Every turn, a single card is drawn randomly from the remaining cards, and placed by the player on an unoccupied cell on the board. The board is evaluated after filling all the spaces. The final score is the sum of the scores of the combinations found in the rows, columns and the longest diagonals. The possible combinations (and their scores) are:

- Pair (10)
- Two pairs (20)
- Three of a kind (40)
- Straight (50)
- Full House (80)
- Full House with pair of ones and triplet of threes (100)
- Four of a kind (160)
- Straight ten through one (150)
- Four ones (200)

Additionally, another ten points are awarded for any combination on each of the longest diagonals.

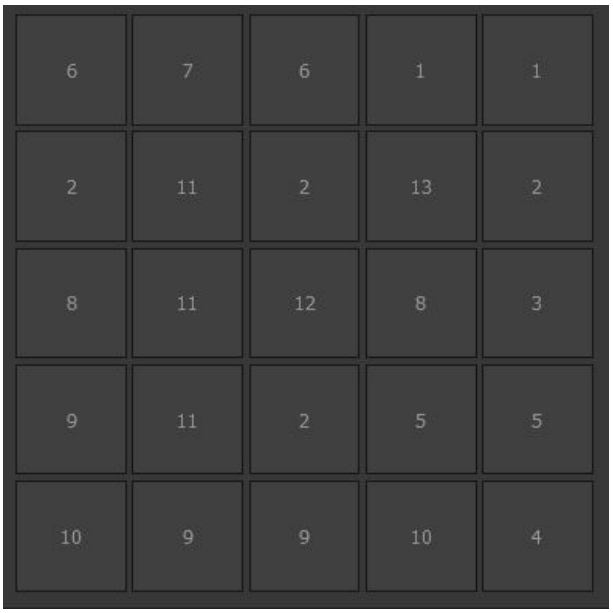

Figure 1: Finished game example

As an example, consider Figure 1. It would be evaluated in the following manner:

- The rows would be evaluated from top to bottom as *Two pairs* (20), *Three of a kind* (40), *Pair* (10), another *Pair* (10) and *Two pairs* (20).
- The columns would be evaluated from left to right as Nothing (0), *Three of a kind* (40), *Pair* (10) and the *Straight* (50).
- The main-diagonal would be counted for nothing (0) and the anti-diagonal would be the *Straight ten through one* with the diagonal bonus (150+10).

Together, that would make 360 points.

For this project, we used the implementation of `Mathematico` that could be found at this Github repository.

4 Algorithmic Approaches

In our work we considered *AlphaZero* to be an expansion of Monte Carlo Tree Search. As the first step, we implemented an MCTS approach with the random-playout roll-out strategy. Then, we added (only) the Value Network on top of the MCTS (as a separate playout strategy), which was trained through the self-play. Finally, we compared our implementation with a customised code from `open_spiel`[2] library. In this chapter, we briefly introduce the details of our implementation.

4.1 Monte Carlo Tree Search

MCTS is a general state-space search algorithm for solving problems with large state spaces. It uses random sampling to approximate a perfect evaluation function. The error of this approximation converges with the growing number of random samples, which makes it suitable for situations with time restrictions (e.g. for making one move).

MCTS works by building a tree of possible game states in four steps:

- **Selection**: A node N of the game tree is selected as an optimal node for further exploration.
- **Expansion**: Creation of new subsequent states based on possible actions from the node N, selected in the previous step.
- **Simulation**: The roll-out policy is used to approximate the (real expected) value of each of the newly created states. The default version of this policy is a random playout, which takes random actions until a leaf state is found. The value of this leaf state is used as an approximation of the state, from which the simulation took place.
- **Backpropagation**: The value used as the approximation is propagated back from the selected node up to the root.

These steps are repeated until the resources run out, which could be either the limit on the number of iterations or the time limit. In the end, the best action from the root (current) state of the game is chosen based on these backpropagated values.

There exist a lot of implementations of MCTS, so we decided against re-implementing it ourselves. We experimented with two implementations. One[3] is fully implemented in Python and is relatively simple. Initially, we chose this one as it enabled us to make changes to the algorithm (the package does not directly support random nodes in the game tree). The other one is implemented in the `open_spiel` library and is more complex but better optimised.

In order to be able to use these libraries, we needed to define `Mathematico` as a Markov decision process. We decided to model each turn in two separate states. The first state without the possibility to choose an action models a random event of drawing the next card from the deck. The second state models a player's action to place a card on an empty cell of the board. We chose this approach to avoid problems where the randomly drawn card got fixed during exploration, which resulted in an incomplete expansion of the state tree.

4.2 *AlphaZero*

AlphaZero is a general reinforcement learning algorithm, designed for perfect information 2-player games. It can be characterised as a cooperation of MCTS with a neural network, which guides the simulation phase.

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

[3] `https://github.com/pbsinclair42/MCTS`

More specifically, consider a game state, in which a player is to make their decision. *AlphaZero* will use the MCTS as described above, with addition that the expansion phase uses a *Value Head* of the network to estimate the expected values of to-be-selected nodes; and the simulation phase uses a *Policy Head* of the network to estimate the promising moves. The roll-out is then a sample based on these estimates.

To obtain good estimates from both heads, during the training, the algorithm is trying to maximise the expected reward. The training consists of repeating these stages:

1. Generate Trajectories[4] using self-play.
2. Teach the network to approximate the MCTS for the collected states.

In this work, we consider the value-head-only based approach, where the roll-out policy selects the state with the highest value; and the implementation of *AlphaZero* by `open_spiel` library, which contains both heads.

4.2.1 Training

Both agents were trained using *Adam* optimiser until the loss converged. The loss functions for the agents during the training were:

1. For the value-head-only implementation, consider the batch of training states $\mathcal{S}$, then the loss is defined as:

$$\mathcal{L}_1 = \sum_{s \in \mathcal{S}} \alpha_s \cdot (mcts(s) - V(s))^2 + \beta_s \cdot (V(s) - score(s))^2$$

   where $mcts(s), V(s), score(s)$ are the value of s approximated by the MCTS, the value network or by the final score of the game which continued from s, respectively. α_s and β_s are scaling factors based on the certainty of the score computed by the MCTS (i.e. a multiple of the normalised logarithm of visited states in the subtree rooted at s (only for α_s), divided by the logarithm of the remaining game length). The motivation behind these factors is to account for uncertainty in the exploration, as typically, the number of MCTS rollouts is much smaller than the number of random choices along any path.

2. The `open_spiel`'s implementation is minimising the quantity:

$$\mathcal{L}_2 = \mathcal{L}_{policy} + \mathcal{L}_{value} + \mathcal{L}_{reg}$$

   where $\mathcal{L}_{policy}$ is a (softmax) categorical cross-entropy between the policy head and the probabilities calculated by MCTS, $\mathcal{L}_{value}$ is a mean squared error between value head and MCTS value and $\mathcal{L}_{reg}$ is a (L_2) regularisation term.

Value-head-only agent was (optionally) pretrained on a generated dataset of random `Mathematico` positions and their (exact) evaluations using the mean squared error.

5 Evaluation

To evaluate the trained agents and compare them with humans, we used these metrics:

- Average Score Over N Games (AVG)
- Tournament Score (TS)
- Average Time Per Game (AVT)

[4] A trajectory is a sequence of states from the initial game state to a final state, with obtained rewards and chosen actions.

5.1 Average Score Over N Games

As `Mathematico` can be viewed as an optimisation game, where each player tries to win by maximising their own score, the AVG represents the expected score a player achieves in one game. For the evaluation, we used a pre-defined set of games $\mathcal{G}$. The average score of player p is then computed as:

$$AVG(p) = \frac{1}{|\mathcal{G}|} \sum_{G \in \mathcal{G}} s_G(p)$$

where $s_G(p)$ is the final score of the player p on the game G.

5.2 Tournament Score

Similar to *ELO* ratings, TS measures how well the players play against each other. The winner of a `Mathematico` game is usually the player with the highest score, i.e. irrelevant of the magnitude of the score difference against other players. In TS, all $|\mathcal{P}| = M$ players play the same set of games $\mathcal{G}$. After each game, the players are rewarded points: the winner is given M points, the second place $M - 1$ points, ..., last player scores 1 point. The points are then summed and the average is taken:

$$TS(p) = \frac{1}{M \cdot |\mathcal{G}|} \sum_{G \in \mathcal{G}} |\{p' \in \mathcal{P} \mid s_G(p) \geq s_G(p')\}|$$

Therefore $TS(p) \in (0, 1]$, and the higher the score, the better the player plays against other opponents. Note that the value is dependent on the choice of $\mathcal{P}$.

5.3 Average Time Per Game

To compare the trained agents with the human play, this metric uses the average wall clock time of one game (consisting of 25 moves).

5.4 Human Agents

To collect the human data on the games $\mathcal{G}$, we have used two different platforms:

- `pygame` interface (as in Figure 1).
- *Google Colab*'s interface[5] written in the Slovak Language based on the target users' preference (see Figure 2).

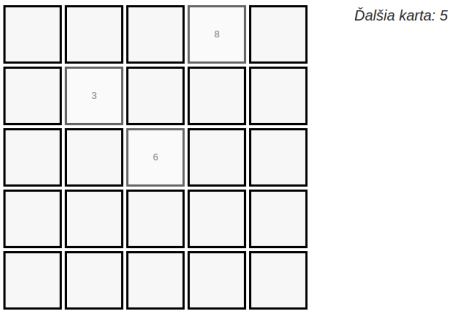

Figure 2: Google Colab's interface. While the numbers might not be very visible, the users reported that they were understandable.

In total, we have collected 4 different responses. Each of the users was then asked about their familiarity with the game `Mathematico`, on a scale 0 – 3. One user marked themselves as a beginner

[5] `https://colab.research.google.com/drive/1PW1nr7JcVvlaiW4_Wvt_TdU1LEFfyuYC?usp=sharing`

(0/3)[6], two users reported a high degree of familiarity (2/3) and one user expert level (3/3). We acknowledge that the number of responses is low and that further evaluation might benefit from a larger sample size.

Player	Average Score	Time Per Game (s)	Tournament Score
human #2 (skill 3/3)	389	117.65	0.85
human #3 (skill 2/3)	356	101.107	0.825
human #1 (skill 2/3)	308	123.751	0.525
human #4 (skill 0/3)	285	163.523	0.4

Table 1: Human performance evaluation on a set of 10 games, for which all evaluators submitted their responses. The tournament scores are calculated by taking into account only human players and the 10 games.

5.5 Results

We have compared different variations of MCTS with random, trained *AlphaZero* agents and the human evaluators – the results for the best (picked) agents are summarised in Table 2[7].

Rank	Player	Average Score	Time per Game (s)
1	human player #2 (skill 3/3)	373	117.65
2	human player #3 (skill 2/3)	366.5	101.12
3	human player #1 (skill 2/3)	303.5	123.75
4	OpenSpiel MCTS [1000]	209.5	66.10
5	MCTS [500]	208	53.07
15	*Mixed* MLP(hidden: 1024) [50]	197	6.95
16	*Mixed* CNN(64x6) [50]	203	49.23
17	MLP(hidden: 1024) [100]	193	14.50
18	MCTS [50]	192.5	4.83
45	random #4	88.5	0.0008
50	OpenSpiel MLP(1024x6) [20]	72.5	1.74

Table 2: Rankings (by TS), Average Scores, and Time of 50 tested players. The [number] in the brackets for MCTS is the number of simulations per move.

Both MCTS implementations achieved similar scores on the evaluation set $\mathcal{G}$, increasing with the number of simulations. We have tested *MCTS* implementation on another set of evaluation games, to examine this property, the results are presented in Figure 3.

[6]This user also played only 50% of the evaluation games with the mean score of 285 – we have therefore excluded them from the comparison of the trained agents.

[7]To see the full table, with the achieved scores for individual games in $\mathcal{G}$, see `evaluation/data.csv` in the attachments.

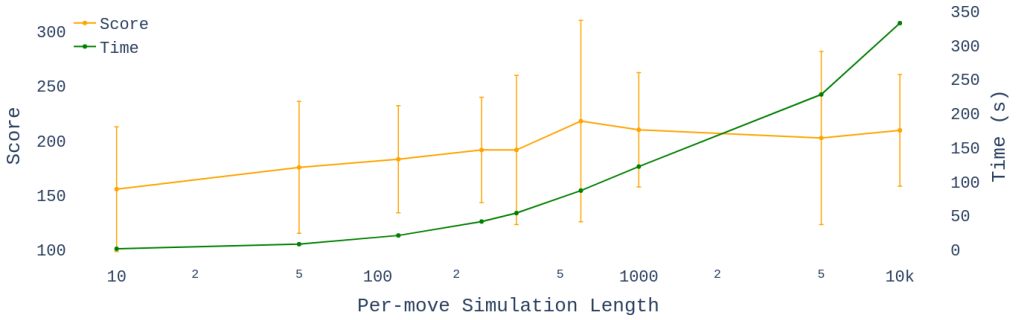

Figure 3: The performance of (pure) MCTS, as a function of the number of iterations per move. While the increase in the number of simulations leads to a logarithmic increase in the strength (the x-axis is logarithmic), the time increases linearly, and even the games with 10K simulations per move do not outperform the humans.

Among different *AlphaZero* agents, no agent achieved better performance than equivalent pure-MCTS agents. Our hypothesis is that the trained neural networks did not have enough approximative power, and we observed, that during training, that (especially smaller models) quickly converged to the constant prediction of the value of a random player. Our time and resources budget did not allow us to train larger networks (the largest had ≈ 18 million parameters and was trained for ≥ 10 hours).

Based on the idea, that neural-network-guided search might be beneficial in the early stages of the game, where it is necessary to run many simulations to faithfully approximate the score, and that the later stages of the game might benefit from more deterministic (pure MCTS) simulations, we have also tested *Mixed* agents. These agents do roll-outs according to the neural network until move 20, from which only random roll-outs are performed. This combination seems to achieve slightly higher scores than the individual constituent agents do. However, to fully verify this idea, it is necessary to conduct further research.

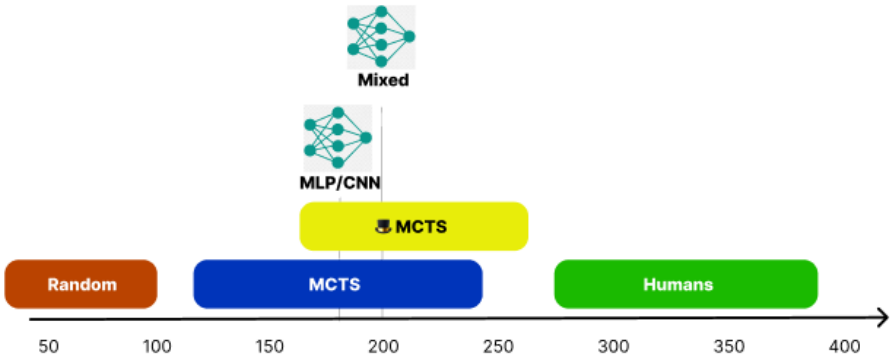

Figure 4: Comparison of analysed approaches. the x-axis is the average score per game. The (orange) random agents score 83 points. The implemented MCTS (blue), based on the number of simulations, is able to achieve average scores up to 250 (for 5,000 simulations per move). The enhanced MCTS implementation (yellow) from `open_spiel` scores in the range 140–250. Humans (green) tend to score around 300 points, on average. Trained *AlphaZero* agents are as strong as MCTS.

Regarding the defined metrics, we have observed strong correlations between ranking by *Average Score* and *Tournament Score* (with significant values of both the Spearman coefficient 0.98 and Kendall's τ 0.92), thus concluding that the scoring methods are (almost) equivalent. Similar observations can be made comparing the ranking of *Average Score* and *Average Time* (0.89 Spearman coefficient, 0.73 Kendall's τ).

6 Conclusion

In the report, we have presented a few agents for solving the game `Mathematico`. We began by introducing the game, and, inspired by the *AlphaZero* paper, we implemented MCTS and *AlphaZero* agents. We have also collected human data and compared the agents.

While all agents confidently defeat the random play, neither of the agents is comparable with human performance, and further research and training might be needed to achieve super-human performance. Trained *AlphaZero* agents were not able to learn the game values, which we hypothesise is due to the restrictions on their size or the training times.

Future work might benefit from a larger sample of human games, and a larger training budget. Other approaches, such as *Deep Q Networks* or even an ensemble of different agents, might also help to achieve better performance.

References

[1] balgot. *Mathematico*. `https://github.com/balgot/mathematico`. 2022.

[2] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. eprint: `arXiv:1712.01815`.

A Documentation

The attached code contains 3 folders:

- `source_code/` with the code for this project;
- `mathematico/` and `azero/` with a copy of (custom) libraries used within this project.

The further instructions in this section assume that you working from within `./source_code/`.

The distributed source code contains multiple files for different purposes:

- To **run the evaluation** and view the **results**, check out `evaluation/`
- To see **examples** and the learning algorithms, **train an agent**, visit `notebooks/`
- To view the implementation, head to `src/`

A.1 Requirements

This project was built using `Python==3.10`, but should be able to support `Python>=3.9 - Python=<3.10`. To install the necessary packages, run:

```
pip install -r requirements.txt
```

A.2 Usage

The section will present the basic usage of the distributed code, for more details, please refer to the documentation within the code.

A.2.1 `Mathematico`

The studied game, `Mathematico`, is part of another package. To install the game, use:

```
URL='git+https://github.com/balgot/mathematico.git#egg=mathematico&subdirectory=game'
pip install $URL
```

In order to play the game, you need to supply a `Player` instance to the `Mathematico` object, e.g.:

```
from mathematico import Mathematico, RandomPlayer

game = Mathematico()
player1 = RandomPlayer()
game.add_player(player1)
game.add_player(RandomPlayer())
# ... add as many players as needed
game.play()
```

which returns the achieved scores per specified players in the order they were added to the game.

For other options of installation (`Python<3.9`), detailed rules explanation, and detailed interface options refer to this GitHub repository. See also `notebooks/mathematico.ipynb` for examples.

A.2.2 `open_spiel` adaptation of `Mathematico`

To use `Mathematico` from the `open_spiel` (`pyspiel`) package, it is sufficient to import one package:

```
import src.agents.ospiel  # registers the game automatically
import pyspiel
```

```
game = pyspiel.load_game("mathematico")
state = game.new_initial_state()
```

Check `notebooks.mcts_open_spiel.ipynb` for examples of how to use this package.

A.2.3 MCTS Agents

There are two types of MCTS agents in this repository:

- Customized Python implementation, inspired by mcts, see the corresponding class for further details:

```
from src.agents.mcts_player import MctsPlayer

MAX_TIME = 500  # 500 ms per move
MAX_SIMULATIONS = 20  # 20 MCTS rollouts per move

custom_mcts_player = MctsPlayer(MAX_TIME, MAX_SIMULATIONS)
```

- `open_spiel` implementations, available as:

```
from src.agents.ospiel import OpenSpielPlayer

MAX_SIMULATIONS = 20  # 20 MCTS rollouts per move

open_spiel_player = OpenSpielPlayer(MAX_SIMULATIONS)
```

A.2.4 Train `open_spiel` AlphaZero agent

To train (customized `open_spiel`) *AlphaZero* agent, use the script at `src/train_azero.py`. This will require authentication for online logging, which you can disable by defining the environment variable `WANDB_MODE=offline`.

```
# To see help
python src/train_azero.py --help

# Train default agent
python src/train_azero.py
```

Loading trained agent Assuming that the previous script saved the configuration and the checkpoints to `PATH/` folder, it is possible to load the trained bot using:

```
from azero import load_trained_bot as _load_azero_bot
import json
import os

PATH = "PATH/"
CHECKPOINT = -1

def load_trained_bot():
    with open(os.path.join(PATH, "config.json"), "r") as f:
        cfg = json.load(f)

    bot, _ = _load_azero_bot(cfg, PATH, CHECKPOINT, is_eval=True)
    return bot
```

The trained bot is not compatible with the `mathematico` interface; therefore, it is necessary to wrap it, for example, by:

```
from src.agents.ospiel import OpenSpielPlayer

# the specific number does not matter here,
# the original (training) value will be used
MAX_SIMULATIONS = 20
player = OpenSpielPlayer(MAX_SIMULATIONS)
player.bot = bot
```

A.2.5 Train (value-network-only) agent

Use notebooks `notebooks/rf-mlp.ipynb` or `notebooks/rf-cnn.ipynb` to train these agents.

A.2.6 Evaluation of trained agents

See `evaluation/perft.py` for detailed instructions and `evaluation/perft.sh` for examples of how to use this script.