Mastering Infexion without Human Knowledge? Probably not...

AI Project Part B

Semester 2 2023

Team Name:
Rolling Raccoons

Team Members:
Aamna Nadeem (1230021)
Hassan Aamer (1235857)

Introduction

Infexion is a perfect information zero-sum game. We explored two different approaches to designing a competent game playing agent. We were inspired by AlphaZero and its ability to learn how to play games effectively without human guidance, and that was the first approach that we used. We then devoted time to developing a Minimax agent with α-β pruning. The details of the two agents, as well as the challenges faced and solutions implemented are discussed below.

Approaches

Agent 1: Simplified AlphaZero

The Simplified AlphaZero algorithm consists of three main components:

1 - The Neural Network:

The AlphaZero agent uses a convolutional neural network (written from scratch using NumPy) with 3 shared layers (2 convolutional & 1 feed forward) and a value and policy head each with its own feedforward layer. The tanH and Sigmoid activation functions are used throughout the neural network. The neural network takes a board state as input and outputs two things, an evaluation of the value [between -1 and 1] of the position as well as a list of move probabilities for the agent to take from this board state. The network is trained with the same loss function that is used in the AlphaZero paper:

$$l = (z - v)^2 - \pi^T log(p) + c||\theta||^2$$

where z is the actual value at game end from the input state, v is the predicted value by the neural network. π is the improved policy after MCTS iterations (discussed below) and p is the predicted policy output. The c term is the regularisation term.

We explored two different optimisers to minimise the cost function:

I. **Stochastic Gradient Descent**
   We used SGD over other gradient descent algorithms like Batch Gradient Descent because it is well suited to deep learning problems and reacts to each training example, which is what we needed. SGD allows us to make the most of our data. We experimented with different values for the two hyper-parameters (the learning rate α and regularisation

coefficient λ) and empirically determined values that would not cause overfitting. The learning rate α was especially tricky as it had to be balanced with the number of training games played (and hence generated) to prevent overfitting or underfitting. We eventually settled on values of 0.01 for α and 1e-5 for λ.

II. Adam

We suspected that our SGD algorithm may not be best suited for the job and decided to explore an alternative called Adam (Adaptive Moment Estimation). This is an extension of SGD that incorporates adaptive learning rates and momentum-like terms to enhance the training process, allowing it to converge faster than SGD. The algorithm maintains two moment estimates, which capture the mean and variance of the gradients. These moment estimates are updated during each iteration of the training process. We also introduced bias corrected terms to account for the initialization biases at the beginning of training. This seemed to provide better results.

2 - (Modified) Monte Carlo Tree Search:

Monte Carlo Tree Search (MCTS) explores the game tree by simulating possible moves and their eventual outcomes. At each step it selects the move which the agent's neural network thinks has the highest probability of winning. It maintains a delicate balance between exploration (simulating unexplored moves) and exploitation (expanding moves that it is pretty confident about) by calculating the UCB score. It then propagates the results up the tree. Eventually, after a certain number of simulations, it selects an action to take from the most promising actions at the root.

Our implementation of AlphaZero however, uses a slightly modified version of MCTS. It is cheaper for the algorithm to simply ask the neural network what it thinks the value of a particular node is instead of running a (possibly long) simulation until the game ends. Therefore, every time the search algorithm expands a node, it gets its value and the prior probability from the neural network, which it then propagates up the tree to eventually compile into a policy vector over the action space from the root, which is what the modified MCTS returns.

3 - Self-play & Reinforcement Learning:

There is no repository of games played by Infexion grandmasters from which our algorithm can take training examples to perform supervised learning. Therefore, our algorithm has the difficult task of first generating training data and then optimising its parameters accordingly. AlphaZero starts with a randomly initialised neural network and plays 25 games against itself using the modified MCTS outlined above to take actions on each round. Each action made is compiled into

1 training example, alongside the board state at which it was taken, and the resulting value of the game (+1 if RED won, 0 if it was a Draw and -1 if BLUE won).

These training examples are then used to train the neural network (see the neural network section for details). The trained neural network then competes against the untrained network for 15 games. If the new neural network can win 60% of these games, then the previous one (and associated training examples) is discarded and the new neural network is used to generate training examples for the next epoch. There are 10 such epochs. However, if the network fails to beat the previous network, then the new network is discarded and the old network is used to generate even more training examples to be used with the previous ones to have another try at training. This process is repeated until the new network wins.

(An example of this is shown in the figure to the right)

```
Head to head, Game 3
50 moves played
100 moves played
150 moves played
200 moves played
250 moves played
300 moves played
RED WON
OLD POWER BLUE: 19
NEW POWER RED: 30
Winner: 1
Old won: 1/7
New won: 3/7
Drawn: 0/7
```

This process gradually improves the neural network's performance by learning from its mistakes and successes. It slowly develops a strategic insight into the game of Infexion and, with enough training time and computation power, can beat even the most sophisticated of computer programs.

Optimizations:

Utilising NumPy Arrays for Mathematical Operations:
To enhance the computational efficiency of mathematical operations, we used NumPy arrays instead of conventional Python lists. NumPy arrays offer efficient memory management and vectorized operations - allowing faster computations.

Utilizing Lists for Dynamic Appending:
NumPy arrays can be less efficient when it comes to dynamic appending of elements. Lists are highly optimised for dynamic resizing, making them more suitable for scenarios where frequent appending of items was required, such as expanding a node to add its children or generating a list of possible moves. This improved performance and reduced memory overhead during the search process.

Problem: The regularisation term dominated the loss function in our first few tries.
Our Solution: So we reduced its effect on the total loss function to be more in-line with the value and policy loss so that the network's primary goal is not to reduce the size of the weights, but to reduce the policy and the value loss. The regularisation term should act as a safeguard, not the primary motivators.

Agent 2: Minimax Implementation:

Our implementation of the Minimax algorithm consists of two main components:

1 - The Minimax Algorithm:

In our implementation, the Minimax algorithm traverses the game tree, considering all possible moves and their outcomes. At each level, it alternates between maximising the score for the current player and minimising the score for the opponent. This process continues until a terminal state is reached, such as a win, loss, or draw. This value is then propagated up the game tree, assuming perfect play from both sides.

However, due to the huge branching factor and game tree depth of Infexion, it is not possible to fully traverse the game tree until a terminal state is reached. Assuming a worst case scenario, Infexion has a theoretical upper bound of 343 for the action space, practically this is about 50 - 100 valid moves at each game state. This, compared with the fact that a game, assuming perfect play, lasts around 100+ moves, leads to a huge game tree.

Therefore, it is necessary to stop at a predetermined depth limit. For our implementation, we **chose a depth of 2 (with the root at depth 0)**. A depth of 3 was violating the time constraints (it would run out of thinking time at complicated board states).

Nodes at this depth are likely not terminal states, and therefore it is necessary to write an evaluation function for the game state to propagate up the tree.

To optimise the algorithm's performance, we implemented alpha-beta pruning, which reduces the number of nodes evaluated by eliminating branches that are guaranteed to be worse than previously examined branches. This technique significantly improves the search efficiency, allowing deeper exploration of the game tree within a reasonable time frame. This technique requires a good move ordering that explores best moves first.

We chose to order the valid moves at any given time by considering spread actions before spawn actions. This is because it is impossible to win with a spawn action, a spread action will, more often than not, result in a positive swing in the evaluation. This ensures that our tree is pruned efficiently.

2 - The Evaluation Function:

We deliberated about a lot of different evaluation functions, from complicated to really simple. We ended up going with a simple evaluation function:

$$V_{red} = \sum_{i \in board} power_{red}(i) + \sum_{i \in board} (i == red)$$

$$V_{blue} = \sum_{i \in board} power_{blue}(i) + \sum_{i \in board} (i == blue)$$

$$EVAL = V_{red} - V_{blue}$$

Essentially, the function considers the power and the number of cells a player has on the board. Although the game ends when the opponent has no cells left, the amount of power a particular player has directly affects its ability to:

- Control large sections of the board, which limits the ability of the opponent to spawn in "uncontrolled" empty spaces *(i.e. empty spaces that cannot be filled by the current player in the next spread action)*
- Spread over large swathes of the board, which can swing a game easily

A lot of other terms were considered but eventually dropped:

- **Average power per cell:** This was not added because often, a spread out opponent is harder to beat because the amount of cells they contain open up a lot of spreading opportunities, and makes it harder for them to get eliminated in one move. A player with one cell with 6 power is at a great disadvantage compared to a player with 6 cells each with 1 power.

- **Enemy cells under attack:** Theoretically, this is a great addition. It reflects the maximum amount of enemy cells that can be captured in one spread action. This allows the player to value positional play over just mindlessly attacking pieces. The practical issue with this is that it is cumbersome to calculate repeatedly and causes the agent to be excessively slow. Also, it is hard to weigh a factor like this, because the agent then tends to prefer positions where there are a lot of cells under attack, but then it doesn't actually take over opponent cells. Which is the point of the game.

- **Empty cells under attack:** The motivation behind this term is to prevent the enemy from spawning in places which cannot be taken in one action. So the agent tries to keep as many empty cells under attack as it can. This term was not included because it is cumbersome to calculate and also suffers from the same problem as the previous term does, that it then tends to overvalue positioning over capturing

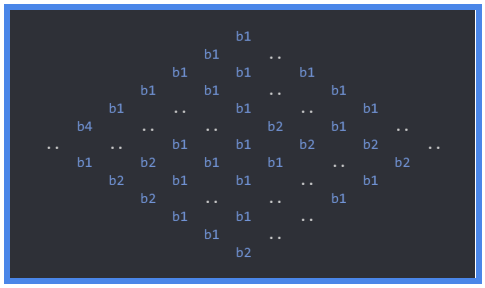
All of the above mentioned techniques often made the game "over-engineer" simple positions that were winning in 3+ moves into a non-existent tactical puzzle, and were hence dropped.

Optimizations:

- Declared a global array of the action space and then stored indexes of actions inside each node in the MCTS (Agent 1) and Minimax (Agent 2) game tree. This significantly reduced the memory utilisation of our algorithms.
- Each evaluated board position in Minimax (Agent 2) is then stored in a dictionary with the state as the key and the evaluation as our value. The buffer size of the dictionary is set to 12,000 in our program as this does not violate space constraints. Any new states added after that cause the oldest states to be discarded. This optimisation causes the computationally expensive evaluation to not be done repeatedly for the same state.

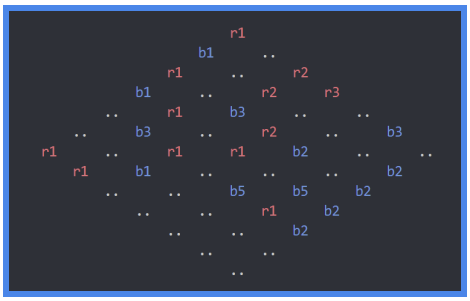
Evaluating Performances and Conclusion

To evaluate and compare the performance of the two agents, we decided to make the two agents compete against each other in a total of 20 games, as well as against a random bot (Agent R) as control, alternating the playing colour of the agents for each game.



Minimax shreds AlphaZero

Results: Agent 1 and Agent R comprehensively lost all games played against Agent 2. Agent 1 exhibited sub-optimal decision-making and failed to adapt its strategy effectively. The reason for this was insufficient training time due to constraints with computational power. We were only able to provide a limited training duration for AlphaZero, which proved to be inadequate to reach its full potential - (near) perfect play.



AlphaZero (Blue) outcompetes Agent R at Move 343... 20 times

Agent 1 did however, win all of its games Agent R, which shows that it is still a great prototype that can be expanded upon.

Therefore, we decided to go with Agent 2 (Minimax) as our final submission.

Sources used:

Huber, A. (2021, June 21). *A 2021 Guide to improving CNNs-Optimizers: Adam vs SGD*. Medium. Retrieved May 14, 2023, from https://medium.com/geekculture/a-2021-guide-to-improving-cnns-optimizers-adam-vs-sgd-495848ac6008

Motani, M., & Tan Chong Min, J. (n.d.). Brick Tic-Tac-Toe: Exploring the Generalizability of AlphaZero to Novel Test Environments.

Nair, S. (2017, December). *A Simple Alpha(Go) Zero Tutorial*. https://web.stanford.edu/~surag/posts/alphazero.html

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Driessche, G. van den, Graepel, T. & Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature*, *550*(7676), 354–359. https://doi.org/10.1038/nature24270