Strategy Selection Based on AI Agents

Gobblet Gobblers as a test case

Matan Kutz **Meitar Basson**

Introduction -

The main goal of our project is to investigate whether simple AI techniques can be used to discover and explain winning strategies for games. For example, in the classic game of Tic-Tac-Toe, a strategy such as controlling the center square and forming diagonal threats is well known. Our aim was not only to find a winning strategy and create a winning agent but also to ensure that the reasoning behind it would be understandable to humans. This contrasts with many modern AI approaches where by having some reasonable heuristic the agent will do the rest and the exact strategy is often opaque.

Additionally, we wanted to provide strategy recommendations that adapt to different stages of a game. For instance, in chess, early moves focus on central control, while mid and late-game strategies vary depending on the board state, number and position of pawns for example could affect the strategy.

To test our approach, we chose the game *Gobblet Gobblers*, a more advanced version of Tic-Tac-Toe (rules here). The game introduces additional complexity through the use of stackable pieces of varying sizes, while still maintaining an accessible rule set. This made it a suitable candidate for exploring human-understandable heuristics and strategies.

Problem Description -

We encountered several challenges early in the project. Our initial idea was to train a sparse neural network, such that each node combines a very limited number of features, which would help explain strategies in a human readable way. To add an aspect of game stages we used Feature-wise Linear Modulation (FiLM) and integrated it into our network. However, training the network required a dataset of play games, which unfortunately we could not find online. While Tac-Tac-Toe datasets are available, they are not applicable due to the increased complexity of Gobblet Gobblers.

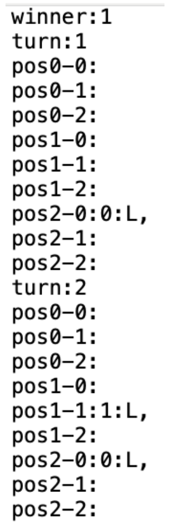
As a result, we decided to generate our own dataset by implementing the game and simulating over 5000 games using heuristics we wrote ourselves. Having our dataset we extracted a list of features that we wanted our network to be based upon and the final heuristic generated by the network was tested against human players, our original heuristics and also ChatGPT.

We kept our game agents to a very basic level, often looking only one turn ahead and following one or two simple rules (e.g. cover rival pieces with your own, use large pieces, block imminent rival victories, etc). This was done in order to keep our effect or the learning process to a minimum, since creating a sophisticated heuristic and training the network on data generated from its games could lead to the network arriving at the same heuristic it was trained on, and so no real learning would occur.

Dataset Creation -

As previously stated, we simulated many games between different AI agents in order to create a dataset. We tried to create several agents, each using another heuristic in order to get some variance in our dataset. Our heuristics were as follows -

1. **Blocking agent** - this agent will win only if it is possible to win within the next move, otherwise it will prefer blocking the opponent or just playing randomly in the case where no block is possible.
2. **2 Steps agent** - the agent will try winning in the next move. In the case where it's not possible to win in the next move, the agent will look ahead for his next move and simulate whether he could win. It is balanced with the opponent's chance of winning in its turn.
3. **Undercover agent** - this agent will win only if it is possible to win within the next move, otherwise it will prefer *covering* the opponent's piece randomly.
4. **Proactive agent** - this agent will win only if it is possible to win within the next move. If not, this agent will deepen its search and determine its best move considering the opponents best options as well.
5. **Blocking only agent** - unlike the previous stated blocking agent, this agent focuses only on blocking the opponent without even trying to win.

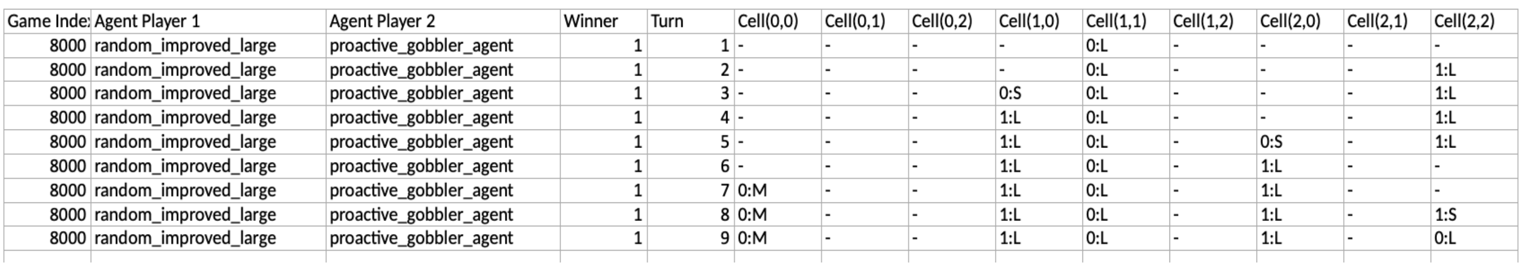


```
winner:1
turn:1
pos0-0:
pos0-1:
pos0-2:
pos1-0:
pos1-1:
pos1-2:
pos2-0:0:L,
pos2-1:
pos2-2:
turn:2
pos0-0:
pos0-1:
pos0-2:
pos1-0:
pos1-1:1:L,
pos1-2:
pos2-0:0:L,
pos2-1:
pos2-2:
```

For each simulation we created a log file (partial example attached) which we used in order to create the dataset excel sheet and also extract the relevant features.

Each board state was encoded using a fixed 3x3 grid structure, with positions ordered from top-left to bottom-right using 0-indexing. For every occupied cell, we recorded both the player ID and piece size (could be plural). For example, the entry pos2-0 refers to the cell in the third row, first column, which contains the first player large piece (0:L).

We also wrote a script log_to_excel in order to create an excel file containing the data in a more readable version. Snippet from this excel file is attached below -



Game Inde	Agent Player 1	Agent Player 2	Winner	Turn	Cell(0,0)	Cell(0,1)	Cell(0,2)	Cell(1,0)	Cell(1,1)	Cell(1,2)	Cell(2,0)	Cell(2,1)	Cell(2,2)
8000	random_improved_large	proactive_gobbler_agent	1	1	-	-	-	-	0:L	-	-	-	-
8000	random_improved_large	proactive_gobbler_agent	1	2	-	-	-	-	0:L	-	-	-	1:L
8000	random_improved_large	proactive_gobbler_agent	1	3	-	-	-	0:S	0:L	-	-	-	1:L
8000	random_improved_large	proactive_gobbler_agent	1	4	-	-	-	1:L	0:L	-	-	-	1:L
8000	random_improved_large	proactive_gobbler_agent	1	5	-	-	-	1:L	0:L	-	0:S	-	1:L
8000	random_improved_large	proactive_gobbler_agent	1	6	-	-	-	1:L	0:L	-	1:L	-	-
8000	random_improved_large	proactive_gobbler_agent	1	7	0:M	-	-	1:L	0:L	-	1:L	-	-
8000	random_improved_large	proactive_gobbler_agent	1	8	0:M	-	-	1:L	0:L	-	1:L	-	1:S
8000	random_improved_large	proactive_gobbler_agent	1	9	0:M	-	-	1:L	0:L	-	1:L	-	0:L

Features Extracted -

In order to train our neural network effectively, we decided to design a set of 25 features that represent each board state in a compact, and informative manner. These features aim to capture both spatial properties of the board and strategic elements relevant to the gameplay.

Below is a summary of the features we used -

1. Column counts - number of the pieces in each column
2. Row counts - number of the pieces in each column
3. Winning/Losing player pieces on board - how many pieces of each size
4. Winning/Losing player pieces captured - how many pieces of each size were captured
5. Average location (vertical / horizontal) of the player's pieces
6. Player spread - how spread out the player's pieces are across the board.
7. Total pieces on board - both players combined
8. Game progress - vector to encode the game stage, early, mid or late.

This feature set was designed to reflect meaningful aspects of the game that could help a neural network (and also a human observer) infer potential strategies. For example:

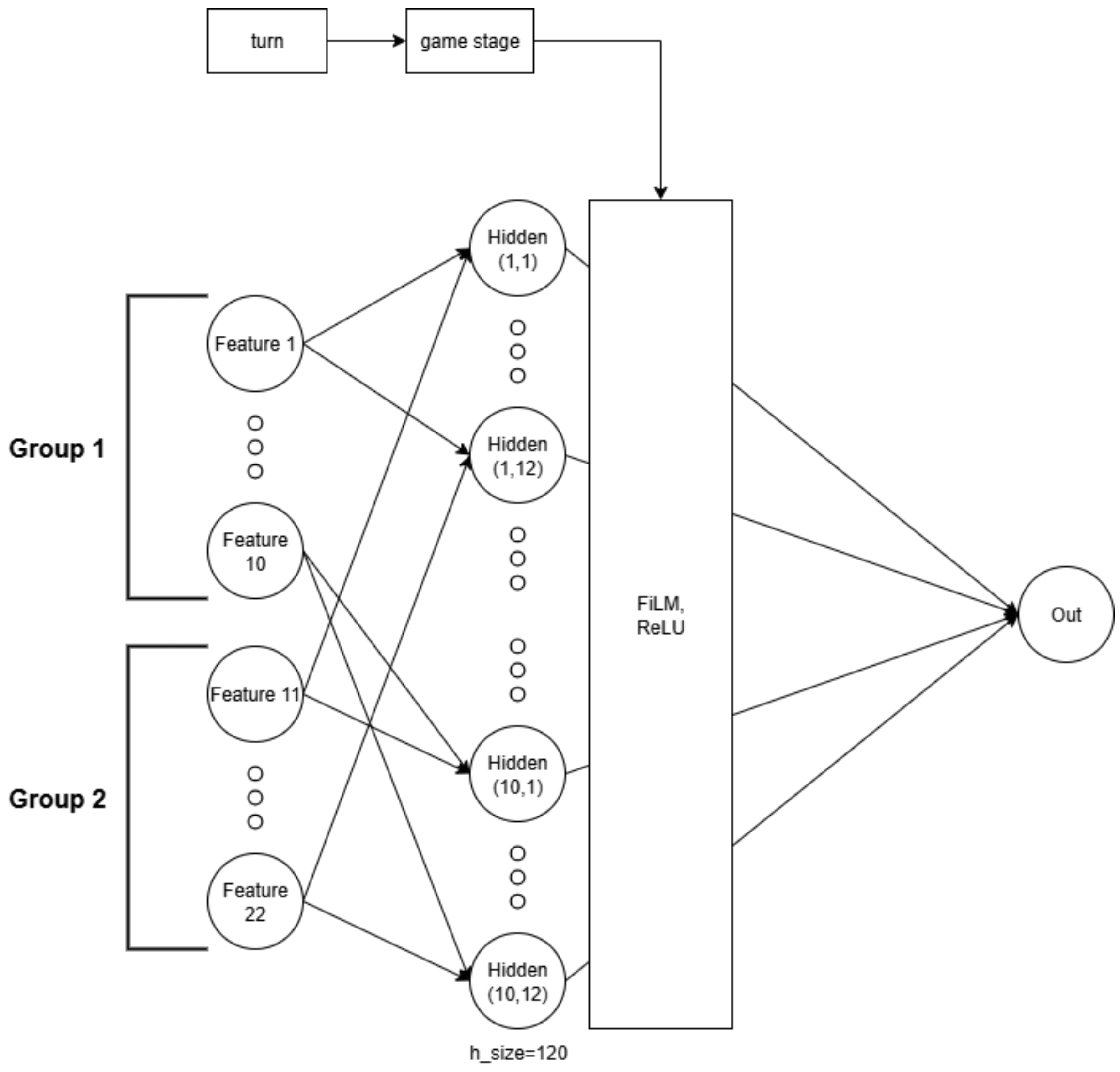
- Column and row counts give insight into imminent threats or opportunities to align three pieces.
- Piece sizes matter since larger pieces can cover smaller ones.
- Spatial features like average position and spread are helpful for understanding board control.
- The game progress is used in order to generate a strategy for each game stage as stated earlier.

Together, these features form the input model to the neural network.

Neural Network Architecture -

The neural network consists of a sparsely connected, 2-layer MLP, integrated with FiLM. The input layer includes 22 nodes, each representing one of the features of a game state, excluding the temporal features. The features are divided into 2 groups, one with a focus on location and one with a focus on quantity and size. This division was chosen in order to create more helpful combinations and avoid collisions between features that might contradict each other. e.g. "place large pieces in the center column" could be helpful, but "place pieces in rows 1 or 2" is not, since these two instructions cannot be followed together, as each piece is placed in one row.

The hidden layer contains 120 nodes, each connected to 2 input nodes, one of each group, such that every possible combination is present. FiLM modulates the network's processing of the data differently according to the game stage



Proposed Strategy -

Besides our main goal of suggesting a strategy for the game, another goal was to develop an AI agent for the game that adapts its strategy dynamically depending on the stage of the game - early, mid or late.

In total, we generated over 39k game states, split into 31k training samples and 8k testing samples by simulating games between various heuristics agents. The network was trained for 50 epochs, achieving a final training loss of approximately 0.156 and test loss of 0.162 , indicating good generalization.

We were using the total number of pieces on board to classify the game into three stages:

- Early game - up to 4 pieces on board
- Mid game - 5 to 8 pieces on board
- Late game - more than 8 pieces on board

For each stage, the agent loads the corresponding set of learned heuristics feature pairs, each associated with weights and modulation parameters (Gamma, Beta). We then tried to prioritize and avoid specific board features and feature combinations depending on the game stage, based on patterns learned from our simulations.

Early Game Strategy -

- Get control of key columns and also place large pieces in central or threatening positions. Control the center if possible.
- Keep the total number of pieces balanced and avoid over concentration in single rows or columns.

Mid Game Strategy -

- Focus on maintaining control of rows and columns where 'two in a row' configurations are possible.

- Pay special attention to the location of our medium and the opponent's small pieces, as these can be used tactically to cover an opponent's piece or block key positions.

Late Game Strategy -

- Continue to control the center and key rows or columns with large pieces to restrict the opponent's options.
- Avoid allowing the opponent to gain leverage by placing pieces in critical positions that can quickly lead to a winning position.

When we first wrote the 'agent' we saw that sometimes he doesn't block the opponent's threats and even blocks his own pieces. We forced the agent manually to avoid those moves and the results were great against most agents. Therefore we believe this strategy would actually help a human agent get a good win rate playing the Gobblet Gobblers game.

Agents Pseudocode -

```
func greedy_improved_agent(state, player, time_limit):
    best_score = -infty
    best_move = None
    for move, next_state in get_neighbors(state):
        score = calc_score_based_on_pieces_and_size(next_state, player)
        if score >= best_score:
            update_score_and_move
    return best_move
```

```
function random_improved_agent(state, player, time_limit):
    winning_move = check_winning_move(state, player)
     if winning_move exists:
          return winning_move
    best_moves = get_best_moves_by_future_win_chances(state, player)
    if best_moves not empty:
        return random choice from best_moves
else:
    return None
```

function blocking_random_agent(state, player, time_limit):

```
    moves = state neighbors
    if no moves:
            return None

    for action, next_state in moves:
            if next_state is winning for player:
                return action

    blocking_moves = get_safe_blocking_moves(state, player, time_limit)
    if time exceeded:
        return random from blocking_moves or moves

    return random from blocking_moves or moves
```

function blocking_more_random_agent(state, player, time_limit):

```
    moves = state neighbors
    if no moves:
        return None
```

```
            blocking_moves = get_safe_blocking_moves(state, player, time_limit)

        if time exceeded:
            return random from blocking_moves or moves
        return random from blocking_moves or moves
```

```
function proactive_gobbler_agent(state, player, time_limit):
    best_score = -infty
    best_actions = []

    for each action, next_state in state neighbors:
        if next_state is winning for player:
            return action
        opponent_moves = next_state neighbors
        if no opponent_moves:
            score = evaluate(next_state, player)
        else:
            score = min(evaluate(opp_state, player) for opp_state in opponent_moves)
        if score > best_score:
            best_score = score
            best_actions = [action]
        else if score == best_score:
            best_actions.append(action)
        if time limit almost reached:
            break

    if best_actions empty:
        return random move

    return random choice from best_actions
```

function stage_adaptive_agent(state, player, time_limit):

```
    neighbors = state.get_neighbors()
    if no neighbors:
        return None

    if winning_move exists:
        return winning_move

    identify recently placed pieces to avoid moving them

    classify moves:
        - blocking_moves: moves that don't allow opponent immediate win
        - new_piece_blocking, existing_piece_blocking
        - center_blocking_moves (prioritize protecting center)

    filter neighbors by priority:
          center & new_piece blocking > new_piece blocking > center & existing_piece blocking >
existing_piece blocking > blocking_moves

    if no filtered neighbors:
        filtered_neighbors = neighbors

    determine game stage by total pieces on board
```

```
load heuristics for that stage

best_score = -infty
best_moves = []

for each filtered neighbor:
    if time exceeded:
        break
    compute score using heuristic
    if winning move found:
        return move
    update best_score and best_moves accordingly

return random choice from best_moves or filtered_neighbors
```

Experiments and Results -

After implementing the new agent, we evaluated its performance by measuring its win rate against several other agents. The opponents were selected from the same set of agents used to generate the log files that informed the neural network used in order to train our agent.

Our so-called *'best'* human perceivable agent played 200 games* against each of the opponents. The results are summarized in the table below and the full game records are provided in the relevant excel sheet attached to the project.

We also tested the agent against ChatGPT and Gemini agents, expecting medium results. Surprisingly, both performed poorly, losing all matches. Interestingly, a human opponent also struggled against the agent, suggesting that a player following the same strategy would achieve excellent results.

* Note that out of the 200 games, the agent played first in exactly half.

Opponent	Best agent winning %	Opponent winning %	Ties %
greedy improved	100	0	0
proactive	50	50	0
blocking random	86.5	11	2.5
blocking more random	89	5.5	5.5
random improved	95.5	4.5	0
ChatGPT/Gemini	100	0	0
Human	55	45	0

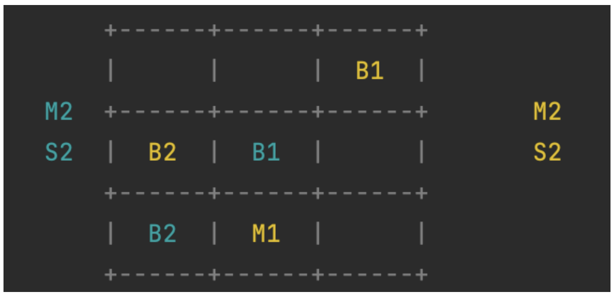
- Greedy Improved - this agent is based on the number of pieces on the board (for the current player) and piece size.
- Proactive - Uses the MiniMax algorithm, considering both piece count and size. This was by far the best opponent our agent had to play against.
- Blocking random - Attempts to win first, if no winning move is available, blocks the opponent, otherwise plays randomly.

- Blocking more random - Attempts to block first, if no blocking move is available, will play randomly.
- Random Improved - Uses MiniMax with a heuristic based on whether a move can lead to an immediate win.
- ChatGPT / Gemini - Performed poorly, losing relatively quickly.

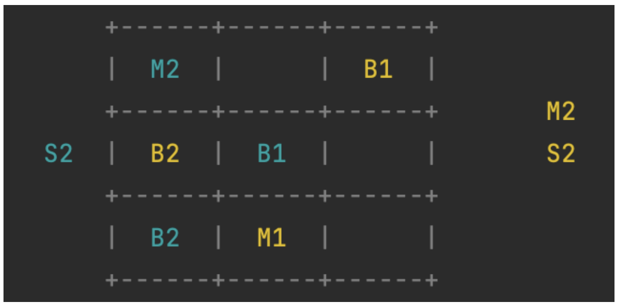
<u>Results analysis</u> -

- Tied games - We examined several matches that ended in a tie and noticed that, in some cases, our agent missed potential winning opportunities. For instance, in game log *15424* (can be found in the attached excel sheet), the agent could have forced a win in two moves during the mid-game. This oversight likely occurred because the agent's search depth is limited and there is no lookahead.

  As illustrated in the attached screenshots, moving a medium-sized piece to position (1,1) would have left the opponent without a favorable response, resulting in a forced loss on the following turn. This situation is also called a zugzwang in chess - a position where the player to move is compelled to make a disadvantageous move.



```
     +------+------+------+
     |      |      |  B1  |
M2   +------+------+------+        M2
S2   |  B2  |  B1  |      |        S2
     +------+------+------+
     |  B2  |  M1  |      |
     +------+------+------+
```



```
     +------+------+------+
     |  M2  |      |  B1  |
     +------+------+------+        M2
S2   |  B2  |  B1  |      |        S2
     +------+------+------+
     |  B2  |  M1  |      |
     +------+------+------+
```

- Our best agent's average move time was under one second, similar to most other agents, indicating that the heuristic is efficient and not computationally expensive.
- Lastly, it is worth noting that the 50% winning rate against the Proactive agent is due to the game being solved: the starting player can always force a win. This explains the evenly split results when both agents alternate as the starting player. Having said that, note that while the Proactive agent relies purely on the MiniMax algorithm, our agent focuses on providing a clear, human-understandable strategy.

<u>Project's Limitations</u> -

- Testing - Although our agent played over 1000 matches and achieved a winning rate of over 84%, fully validating its effectiveness would require testing a broader range of opponents, including additional AI agents and diverse human players.
- Human testing - For a computer, calculating the relevant factors to determine the optimal move based on the suggested strategy is straightforward. For human players, however, this process is significantly more challenging and would likely result in a lower win rate. Also different human players would probably play differently and have a broad range of win rate. On the other hand, human players tend to think more and could analyze the board more and find possible winning moves that the agent missed as mentioned earlier.
- Project Strategy - our strategy of building a heuristic based on a dataset and neural network might have worked well for our test case but the Gobblet Gobblers game is limited in its different possible boards. It is possible that this strategy won't generalize well for complex games such as chess and possibly even Connect-4. During development, we also introduced 2 strategic rules not present in the dataset, such as controlling the center with a large piece and blocking an opponent's immediate win. Since our dataset was generated from agents that didn't prioritize center control, this strategy did not naturally emerge from the learned heuristic. Similarly, there was no explicit 'blocking' feature, so we manually appended it. For more complex games, the 'base rule set' would likely need to be far richer, with many more assumptions explicitly encoded before training. For example in chess, we would like to control the center using a pawn while moving our knights and bishops forward as well. We might also want to consider performing castling as soon as possible and many more special techniques chess players have to keep in mind while playing.

Future Work -

During our work several different ideas and improvements came down to our mind -

- Extending the approach to other games of similar complexity - In order to verify that our method of strategy proposing really works well it is needed to have more study cases. Possible games could be Connect-4, Hare and Hounds.
- A deeper level of testing should be implemented as well, probably against a large variety of different agents, AI and humans in order to make sure that our agent really reaches above 80% win rate.
- Extending the approach to other more complex games. Even though our approach might work for the same level of complexity games, scaling our strategy to more complex games might break it. It is worth trying to come up with a set of rules where our approach would work and scale well.
- Automating feature extraction instead of manual engineering - in addition to the built heuristic, we also had to add a couple of rules in order to get good results. It is worth trying to get the same result without having to manually encode those rules.

Where AI was used during this project -

Part of the code provided in our project was written by the staff of Intro to ML 236501 and was used here to accelerate things. An example could be the game environment functions. Also we used Matan's homework code in order to have some base heuristics and agents to be used in the project.

Having said that, part of the code used in order to create the log files, save results to excel and also parts of the neural network were written using AI assistance.

We also used AI in order to explore different methods and ideas of how to implement our initial idea. That's how we learned about FiLM for instance.

Summary -

In this project, we explored how AI techniques combined with interpretable heuristics can be used to discover, explain and adapt winning strategies in the game Gobblet Gobblers. Unlike many other models that are usually more black-box style, our approach prioritizes human understandable strategies that dynamically adjust according to different game stages.

We developed multiple heuristic agents and generated a custom dataset by simulating thousands of games. Using this data, we trained a sparsely connected neural network integrated with FiLM to capture stage dependent strategic patterns that could be human understandable.

After experimenting with the suggested agent, the results demonstrate that it consistently outperforms various baseline agents and even human opponents, while maintaining efficient computation. The project suggests that it might be possible to learn game strategic using simple AI tools.