Solving Connect-4 with Artificial Intelligence

CS7IS2 Project (2021/2022)

Mark Hanrahan, Mayank Arora, Claudia Alonso, Aditya Shrivastava

hanrahma@tcd.ie, aroram@tcd.ie, alonsovc@tcd.ie, ashrivas@tcd.ie

Abstract. This work presents several artificially intelligent approaches to solving connect 4. Connect 4 is an adversarial, zero-sum game with a small action state and comparatively massive state space. Due to this large state space, this problem is challenging for AI to solve. This problem is also reflected in many real-world problems concerning control. An example of this is invasive species control [Bogich and Shea, 2008] and recommender systems in smartphones [Dulac-Arnold et al., 2019].

Keywords: artificial intelligence, minimax, Q-learning, neural networks

1 Introduction

This work aims to develop artificially intelligent agents for the game "Connect-4". While deceptively simple, the game "Connect-4" is a complex problem for AI to solve. It is a competitive game in which two players take turns dropping coloured discs into a grid. A player may only drop a disc into a column, the disc falls to the lowest row of that column. The player that connects four discs wins. The discs can be connected in either vertical columns, horizontal rows, or diagonal spaces.

It is essential to develop AI approaches to this game as it is demonstrable of a game with a small action state and exponentially large state space. This is an issue relevant to many real-life problems, such as control of invasive species [Bogich and Shea, 2008]. The action state refers to the option of available columns a player can drop a disc. The state-space refers to the possible number of permutations of discs in the grid. A game with such a large number of possible states can congest and defeat any standard AI approach that does not account for this space. Therefore it is important to identify customisations and optimisations to an artificially intelligent approach to solve this problem.

Link to video:[Link to video presentation]

2 Related Work

2.1 BFS/DFS

Firstly, the classic breadth-first search (BFS) and depth-first search (DFS) algorithms were selected to demonstrate a baseline for this work. These algorithms are considered greedy, with little forward planning.

2.2 Work on Minimax

"Minimax" was selected for a standard adversarial approach. Connect-4 is a zero-sum game, and as such, an algorithm that specifically accounts for this is strategic. Minimax includes an element of forward-planning which is beneficial, however, as this problem includes a large state space this approach becomes unfeasible by memory and processing cost. Alpha-beta pruning may help with this but it is not a reliable solution as the order of possible moves may never be pruned. Combining this with an element of deep learning [du Plessis, 2009] is evidenced to act as a viable approach in large state spaces. This example demonstrated in [du Plessis, 2009] was a large game of tic-tac-toe. That approach utilised the idea of reducing both memory and processing demands of Minimax through combining it with a Neural Network implementation. That approach is synthesized in this work also.

2.3 Work on Q-Learning

Q-learning was chosen as an approach because it is a Reinforced Learning (RL) technique. Q-learning trains with the game and learns the best actions to take to win the game when playing against an array of opponents. In the paper: [Alderton et al., 2019], SARSA and Q-learning were chosen as agents to play Connect 4. The agents were trained against themselves and against each other. Each time an agent played against another it was called a 'Test'. Each Test ran for 1000 iterations. However, it does not seem as though the q-tables were saved. Therefore, each time the agent played against someone the q-table was created from scratch. The paper found that which ever player began first gained a big advantage against its opponent. They also found that their exploration rate had a negative correlation with their success rate. This is due to the agent taking more random moves to "explore" the game board. The reward variables were varied between tests but no reward system emerged that had any great success. From this paper, it was concluded that the Q-learning algorithm used by the researchers should:

- Maximise its success rate by starting in the centre column and bottom row whenever possible
- Keep a record of the Q-table to reuse when playing new opponents on the same board
- To have a Q-table for when learning is performed as player one and another for when learning is performed as player two

[Dulac-Arnold et al., 2019] highlights the challenges of real-world reinforcement learning. One of the challenges mentioned in [Dulac-Arnold et al., 2019] is the presence of multi-objective rewards in a real world environment. For a multi-objective environment, the global reward function can be defined as a linear combination or sum of sub-rewards r(s,a) = $\sum_{j=1}^{K} r_j(s, a)$ [Dulac-Arnold et al., 2019]. r(s, a) is the reward for a state-action pair in the learning environment. Connect 4 is a problem consisting of multi-objective rewards. An AI

agent solving connect four can be sequentially rewarded for placing two, three and four pieces in a row during its turn. The reward for placing four pieces in a row can be modelled by taking the sum of placing 2 & 3 pieces in a row. Similarly, an RL agent playing connect 4 can be rewarded for preventing its opponent from placing 4, 3 or 2 pieces in a row. If an RL agent playing connect 4 against humans was to be deployed as a production system, it would be important to have the agent train on a reward function that is built accurately. In the current implementation, the researchers have modelled the reward function to reward the Q-learning agent if it prevents the opponent from placing 4 pieces in a row. Modelling sequential rewards for an RL agent placing 2 or 3 pieces in a row was complex, and the team did not implement it due to computational and time constraints. Based on [Dulac-Arnold et al., 2019] the team decided to implement a multi-objective reward function that rewards an agent for winning, drawing and losing a game of connect 4 and preventing a possible move that would allow the opponent to connect four pieces in a row.

3 Problem Definition and Algorithm

3.1 Task Environment

The problem this research builds to address is solving "Connect-4". Before designing a set of possible agents to solve this problem, the Task Environment must first be defined. The environment of this game is fully-observable, meaning each player can view the entire state of the board. It is a competitive, multi-agent environment with two players competing for victory. The actions in this game create deterministic results; while the row cannot be specified, the effects of gravity combined with the current board state create deterministic results. The environment is sequential as an element of forward planning is necessary for effective strategies. The environment is static as it does not change while the agent is deciding which move to take. The game is discrete as the board has a finite number of states, and agents a finite number of percepts. As the distinction between known and unknown refers to the perspective of the agent, both known and unknown natures will be explored in this work.

Task Environment					
Deterministic Fully Observable	Multi-Agent	Deterministic	Sequential	Static	Discrete

Table 1: Task Environment of Connect-4.

For this work, the task environment was simulated using a python script. The implementation used the pygame module[citation for pygame] for game displaying and user interaction. Several helper methods were created, such as viewing the board state, simulating sensors and performing a move, simulating actuators.

3.2 BFS/DFS

To create a baseline for testing different algorithms for connect 4, Breadth-First Search (BFS) and Depth First Search (DFS) were implemented. For BFS, the initial valid moves for a game state were determined, marked as explored and stored in a queue. A position was popped from the queue, and a piece was dropped for that position. If the game state after dropping the piece was a win, the position was iterated upon in explored dictionary until the parent node was found. The column of the parent position was returned as the best move. If the position was not a winning position, all the valid moves in that position were en-queued. If no winning moves were found, the first valid move in the position was returned. During this process of making moves and checking for a win, the game state was copied and reset after testing the branch for that move. For DFS, the queue was replaced by a stack, and the rest of the algorithm was similar to BFS.

3.3 Minimax

Minimax is an adversarial search algorithm that looks ahead a certain number of plays and evaluates the outcomes of possible decisions. These evaluations are maximised for the player decisions and minimised for the opponent's decisions. Evaluations get back-propagated up the tree and inform the next decision the Minimax agent makes. This ensures that the agent picks a move that maximises their chance of playing well and minimises their opponent's chance of winning. Thus ensuring a minimised chance of loss for the agent. Minimax is a traditional algorithm [Dimand and Dimand, 1992]. It achieves the best results in any two-player game zero-sum game with a definite endpoint. The algorithm picks optimal moves that maximise its chance of winning; it does not learn anything. This means that when Minimax plays against an opponent that does not look ahead, i.e BFS or DFS, it can end up losing. This is because the Minimax agent expects its opponent to play moves that maximise their chance of winning. If the game is pictured as a tree, each level contains nodes representing the player's possible moves or the opponent. The children of each node are the possible moves of the next player on a new level. Minimax uses the `search depth` parameter passed to it to know how many levels to look ahead and evaluate. The nodes found at the search depth are terminal nodes. The search depth of Minimax affects the time complexity of the algorithm. Minimax evaluates each level of the search depth. Consequently, the time complexity of the Minimax algorithm is proportional to the size of the search depth variable it is given.

```
function minimax(game_state, depth, maximizing_player):
    if depth = = 0 OR game_over:
        check for winner

    else:
        if maximizing_player:
```

```
            for row, column in get_valid_move():
                game_state.board[row][col] = self.agent_number
                score += self.minimax(game_state, depth-1, False)
                max_eval = max(max_eval,score)
            return max_eval

        else:
            for row, column in get_valid_move():
                game_state.board[row][col] = self.agent_number
                score += self.minimax(game_state, depth-1, True)
                max_eval = min(min_eval,score)
            return max_eval
```

Listing 1.1: Minimax Algorithm Pseudocode

3.4 Q-Learning

Q-learning is a model-free off-policy Reinforcement Learning (RL) Algorithm. An RL agent implementing the Q-learning algorithm was developed and trained to play the game of connect 4 and connect 3. The researchers chose an RL algorithm to observe the efficiency and effectiveness of Q-learning against a problem with a small action space and a large state space. The researched also wanted to find optimisations within the Q-learning algorithm that would help the RL agent have a higher win rate at connect 3 and connect 4 when playing against other agents or humans.

The Q-learning algorithm implemented by the team to solve the problem of connect 4 and connect 3 can be seen in equations (1) & (2) below.

$$Q(s,a) = (1-\alpha)Q(s,a) + (\alpha)[sample] \tag{1}$$

$$sample = R(s,a,s') + \gamma \max_{\mathbf{a'}} Q(s,a) \tag{2}$$

Equations (1) & (2) represent sample-based Q-value iteration. Q(s,a) is the Q-value for a state action pair. In the current implementation, all Q-values are initialized to 0 at the start. α represents the learning rate. α is set between 0 & 1. $\alpha = 0$ implies that the Q value is never updated and hence nothing new is learnt whereas $\alpha = 0.85$ implies that the algorithm weighs learning from newer experiences higher when compared to previous learnt Q-values.
R(s,a,s') is the reward function. It is the reward the agent receives when it takes an action a' to transition from state s to s'. gamma is the discount factor. max-action Q(s,a) is the Q value for the best action available to the agent in the present state.

The description of the algorithm implemented is as follows:

1. An empty Q-Table is initialised. The Q-table stores the Q-value for each state-action pair. The current state of the board is represented as a two-dimensional array. An action is a column number where the AI player's piece can be dropped.
2. When it is the Q-learning agent's turn to play, the agent either takes a random action with a probability of ϵ or tries to find the best possible valid action to win the game by implementing equation (1) with a probability of 1-ϵ. ϵ is a parameter that helps us set the level of exploration and exploitation for our agent.
3. The best action for the current game state that the agent is in, is calculated by finding the action with the highest Q value for the current state-action pair in the Q table.
4. The Q-learning agent takes the best action in its turn and then updates the q-table for the action it took in the current state. If the Q-learning agent wins the game by taking the action it performed, it receives a positive reward (e.g. R=+2). If the Q-learning agent loses the game by taking the action it performed, it receives a negative reward (e.g. R= -2). If the Q-learning agent ties the game by taking the action it performed, it receives a small positive reward (e.g. R= +0.5).

The performance of the Q-learning agent in the connect 4 and connect 3 environments depends on how various parameters are implemented and tuned. The parameters are as follows:

1. The reward function R(s,a,s'): Tuning the reward function determines how the Q-learning agent performs. If the reward function is not tuned appropriately to the learning environment, the agent's performance can lead to poor results. For example, a higher reward for losing over winning for an AI agent will result in an agent mostly losing against other players. In the current implementation, the Q-learning agent receives a reward of +2 for winning a game, a reward of -2 if it loses the game, and a reward of 0.5 for a tied game. The team made an optimization where the agent receives a reward of +1.5 if the agent prevents the opponent from winning.
2. ϵ-greedy: ϵ helps us communicate to the agent if we want it to prefer exploring the learning environment over exploiting the best policy or vice versa. In our implementation, we start with a high ϵ value so that the agent explores more, and then ϵ is gradually decayed to allow the agent to exploit the best policy.
3. α - The learning rate: α helps us determine how much impact a current sample (equation 2) will have on the present Q value. In the present implementation $\alpha = 0.3$.
4. γ - The discount factor: Adjusting γ can help tune the agent to prefer long term or short term rewards. The current implementation was limited to rewarding the agent only at terminal states, i.e. the end of the game, since the team was time-constrained to implement the logic for determining rewards for each state.

The Q-learning algorithm was tested on connect 3 and connect 4 because connect 4 has a larger state space than connect 3, making testing connect 4 com-

putationally expensive. Connect 4 has 3^{42} possible states (6 rows X 7 columns) over connect 3, which has 3^{20} possible states. (4 rows, 5 columns) [Alderton et al., 2019]

3.5 Deep Learning Agents

A supervised deep learning approach was implemented where several Fully Connected Neural Net (FCNN) models were trained on the moves that led to a win against a random agent. The winning moves were made by BFS, DFS and Minimax with depth set to 8. The model's input was the game board, and the label was the winning move.

For connect 3, the input layer of the neural network model was 20 neurons, representing an entire 4x5 board. The model consisted of 2 hidden layers, having 15 and 10 neurons respectively, and the number of output neurons were set to 5, representing the five columns as valid moves. As choosing a column could be seen as a classification problem, the categorical cross-entropy loss function was chosen. The output label was the best move chosen by an agent, where the agent could be BFS, DFS or minimax.

Initially, separate models were created to test out the neural representation of an agent against itself. The number of training examples were set to 25 thousand, and the number of epochs depended on where the accuracy of the model seemed to plateau. After that, a hybrid model was created that consisted of the winning moves of all the different agents. A similar approach as taken for connect 4, but due to the time constraints to generate winning moves for minimax, this model was trained on seven thousand samples of data.

4 Experimental Results

This chapter details the evaluation process for these AI agents. It covers an outline of methodology, display of results as well as a discussion of these results.

4.1 Methodology

To evaluate all implemented algorithms, a standard framework was created to test and compare their results. The framework tested the AI agents in a connect-4 (6 rows, 7 columns, connect 4) and a connect-3 (4 rows, 5 columns, connect 3) configurations. Connect 4 has a state space of less than $3^{(6*7)}$, while Connect 3 has a state space of less than $3^{(4*5)}$. The smaller state space of connect 3 should benefit the algorithms. These numbers represent upper bounds as disc placement is affected by gravity.

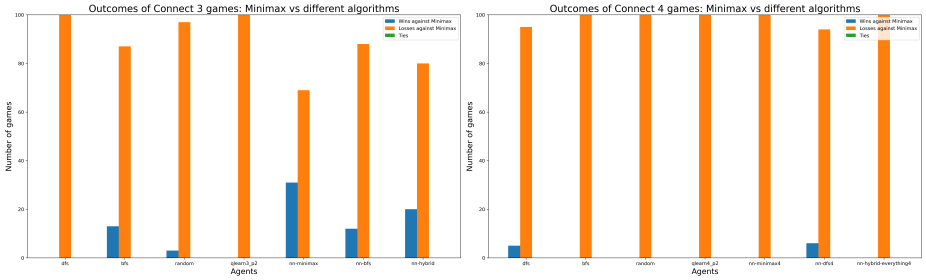
An "Arena" was created to contest each AI agent against the other in a set of 100 games. In this arena, agents were contained in classes defined by a standard interface for "plug-and-play" functionality. In each Arena, one agent was fixed as the first player one and the opponent was rotated between all other AI agents for 100 games each. The first move by each of the players was random, which

helped in breaking determinism in the game. The results of these games (win, lose, tie) were stored in a CSV file. This file was then supplied to our plotting script to show the relative performance against a fixed agent.

The evaluation criteria for this arena were simply the outcome; win, lose and tie. This implementation was not concerned with how much each AI agent won or how long the agent took to compute their next move.

4.2 Results

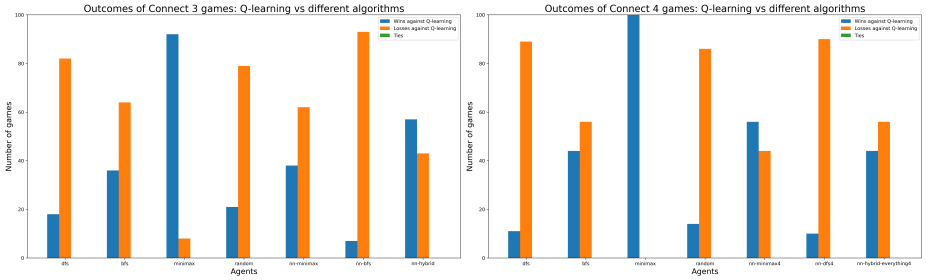
The results of these experiments are shown in the figure 1, figure 2 and figures 3.



(a) Minimax vs different algorithms in Connect 3 environment

(b) Minimax vs different algorithms in Connect 4 environment

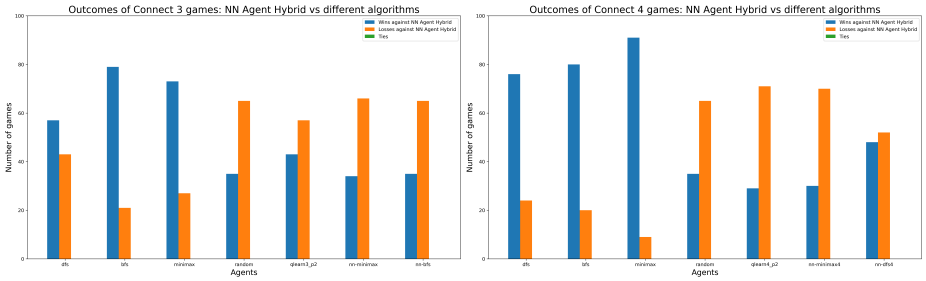
Fig. 1: Result of games played by different algorithms against minimax in a connect 3 and connect 4 environment.



(a) Q-Learning vs different algorithms in Connect 3 environment

(b) Q-Learning vs different algorithms in Connect 4 environment

Fig. 2: Result of games played by different algorithms against Q-learning in a connect 3 and connect 4 environment.



(a) Hybrid neural net vs different algorithms in Connect 3 environment

(b) Hybrid neural net vs different algorithms in Connect 4 environment

Fig. 3: Result of games played by different algorithms against a hybrid neural network agent trained on winning moves of BFS, DFS and Minimax in a connect 3 and connect 4 environment.

4.3 Discussion

For Connect 3 and Connect 4, it was observed that if the minimax agent went as the first player, it performed far better than other algorithms, winning all the games against BFS, Random agent, Q-learing and other neural networks algorithms as seen in Figure 1. Minimax outperforms every algorithm as it plays the first move.

From figure 2 we can observe that Q-learning did not perform well against minimax in connect 3 and connect 4. Q-learning was a close competitor of neural networks trained on the correct moves of minimax with a depth of 8. The neural network trained on the correct moves of minimax was expected to perform better but it did not since the agent was not trained on enough data. Q-learning performs similarly against the neural network trained on correct data from all other agents (other than Q-learning) in connect 3 and connect 4. Q-learning has a high win rate against bfs, dfs and the random agent. In a larger state space Q-learning generally wins more against bfs,dfs and the random agent.

From figure 2 we can see that q-learning and hybrid-nn-everything have a close competition between them. To determine the perfect win-loss rate for closely tied agents we would have to test the agents on several more iterations. Q-learning could perform much better but did not reach its full potential as it was only trained on 20,000 iterations. The performance of the q-learning agent can be vastly improved by training it over more iterations since it would lead to the q-learning discovering more state-action pairs and q-values for these state-action pairs.

As neural networks agents were trained to on data from second move, they were usually at a disadvanctage which an clearly be seen against DFS, BFS and minimax in connect 3/4. against a random, Q-learning and other neural net

approaches it performed better. Every algorithm was consistely able to beat the randon algorithm, especially in a bigger search space of connect 4

5 Conclusions

On implementing, testing and evaluating the aforementioned approaches on a connect 4 and connect 3 environment, the researchers were able to observe that minimax is an effective approach, however it does not scale even with alpha-beta pruning and a depth limiter.
With more time and processing power, the Q-learning algorithm and Hybrid Neural Net could be trained more effectively. With more iterations, the Q-learning agent would have discovered more state-action pairs which would lead to better win-rates by the agent. However, discovering more state-action pairs would lead to larger Q-tables and a very high computational requirement.
With better quality and more numerous and correct move data the Hybrid, deep Neural Net would experience more games and isolate more correct moves as features of the dataset. Solving the connect 4 with the implemented approaches gives us insight into how problems with high state spaces can be tackled with the help of AI agents. To algorithms implemented

References

Alderton et al., 2019. Alderton, E., Wopat, E., and Koffman, J. (2019). Reinforcement learning for connect four. *Reinforcement Learning for Connect Four*.

Bogich and Shea, 2008. Bogich, T. and Shea, K. (2008). A state-dependent model for the optimal management of an invasive metapopulation. *Ecological Applications*, 18(3):748–761.

Dimand and Dimand, 1992. Dimand, R. W. and Dimand, M. (1992). *The Early History of the Theory of Strategic Games*, volume 24. Duke University Press.

du Plessis, 2009. du Plessis, M. C. (2009). A hybrid neural network and minimax algorithm for zero-sum games. In *Proceedings of the 2009 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '09, page 54–59, New York, NY, USA. Association for Computing Machinery.

Dulac-Arnold et al., 2019. Dulac-Arnold, G., Mankowitz, D., and Hester, T. (2019). Challenges of real-world reinforcement learning. *arXiv preprint arXiv:1904.12901*.