Connect N:
A Reinforcement Learning Approach

Lucy Cheng, Angela Fan, Andre Nguyen

1 Problem Description

The problem we chose to work on is that of the classic children's game, Connect-4, but extrapolated to be "Connect-N", where players strive to create streaks of length N with their tokens, while stopping their opponent from creating N-length streaks.

2 Simulator and Driver Function Descriptions

The **simulator** takes two inputs:

1. **n**, the number of tokens that must be connected by a player to win
2. **grid_size**, the height of the square board that will be created by the simulator

The simulator initializes a nested NumPy array of zeroes, where each nested array is a row of the board. Zero represents an empty state. Players 1 and -1 can tell the simulator to make a move by calling the simulator's move method. The board state then adds a 1 or -1 (depending on the player) in the correct spot. When the move method is called, the simulator automatically checks if a player has won.

The move function returns:

1. 1 if the current player making a move has won
2. -1 if the current player making a move is trying to make an illegal move (for example: column does not exist, column is already full)
3. 0 otherwise

and a reward:

1. 50 if the current player making a move has won
2. 0 otherwise

The simulator has a number of helper methods as well:

1. **simulate_move**, which tests to see if a move is valid and returns 1 if not valid
2. **turn**, which returns whose turn it is

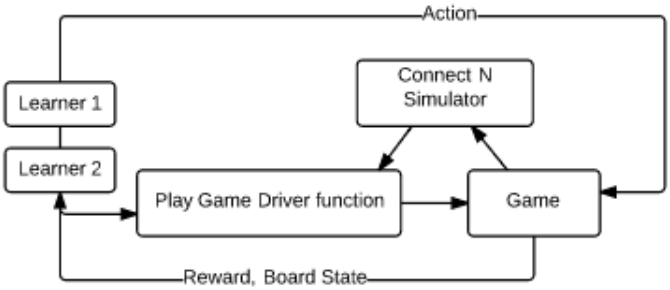


Figure 1: Play Game Driver Function

3. **next_possible_moves**, which returns an array of all possible columns available for a next move (i.e. eliminates columns that are full)
4. **all_tokens_placed**, which returns the location as a (column, row) tuple of all of the tokens that have been placed
5. **is_empty**- takes in a location and returns True if the location is empty and False otherwise
6. **streak** functions that check if a given player has a horizontal, vertical, or diagonal streak going
7. **print_grid**, which prints the current board state

We have written a separate function, called **play_game**, that serves as a driver function to run the Connect N game (diagrammed in **Figure 1**). The driver function takes as input two learners, and uses the Connect N simulator to create a game. The game asks each learner for an action and takes the specified moves. If neither player has won, it continues the game by passing reward and the new board state back to the learners.

3 State and Action Spaces

Possible **actions** are integers representing the numerical column number that a player wishes to put their token into.

The **states** are boards, represented as NumPy arrays. The board is initialized to be full of zeros, which represent empty token spots. Player 1's tokens replace the 0 with 1, and player -1's tokens replace the 0 with -1.

4 Approaches

4.0.1 Baselines

We implemented three baselines:

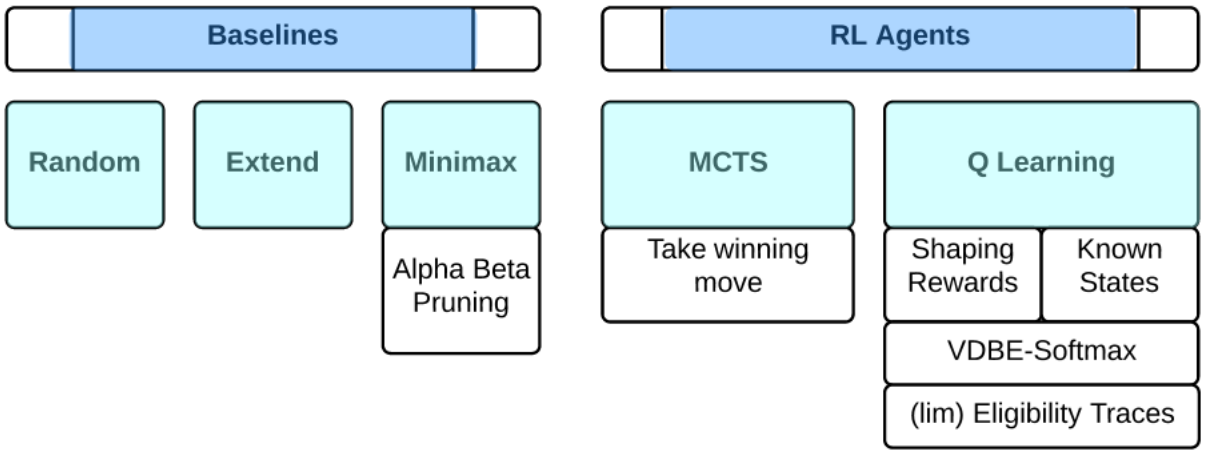


Figure 2: Implemented Baselines and Learners, with Extensions

1. **Random action**, where the agent randomly chooses a column to put its token into at each time step

2. **Extend**, where the agent attempts to extend its largest current streak (horizontally, vertically, or diagonally), breaking ties between streaks randomly.

3. **Minimax**— We implemented the Artificial Intelligence algorithm Minimax, which conceptualizes the Connect-N game as a tree of possible future game states. The current state of the game is the root node of the tree, and each of the children represent possible states resulting from a move that can be made from the current game state. The leaves are the final states of the game, or states where no further moves can be made because one player has one or lost. In traditional Minimax, leaves have a value of either positive infinity (win), negative infinity (lose), or 0 (tie), but such an algorithm forces Minimax to continue evaluating future board states until it reaches a win or a lose. However, this is often computationally infeasible as there are so many possible states in the game. Instead, we implement a **static evaluation** form of Minimax that takes a **depth** parameter that governs how far down the tree the algorithm should explore. We generate node values in Minimax according to this heuristic:

   - Each streak contributes a value of $x * 2^x/2$, where x is the length of the streak, except when $x = n$, where the streak contributes a value of 2000000.
   - Each opponent streak contributes a value $-(x * 2^x)/6$, where x is the length of the streak, except when $x = n$, where the streak contributes a value of -1000000.

   The motivation for these values was that longer streaks are closer to the goal state, and therefore should have much higher value. We chose 2^{x-1} at first, but found the

agent prioritizing creating multiple streaks of length x instead of creating a streak of $x-1$, so we increased the difference in value between streaks even more. At first, we did not have negative values for opponent streaks, which resulted in very boring games between Minimax agents, since there was little interaction between the agents, and the first player always won. The negative value for opponent streaks was added to incentivize agents to try to block the progress of their opponent. We divided by an extra factor of 3 so agents would still prioritize extending their own streak over blocking an opponent streak of the same length. The values for $x = n$ are chosen to represent ∞ since they represent winning or losing the game.

We also explore an extension to Minimax, called **alpha-beta pruning**, motivated by the idea that often Minimax explores more branches than is required. For example, if it needs to expand down to depth 3, but is currently at depth 2, it will continue expanding all branches to depth 3 even if one branch clearly would lead to a loss. In alpha-beta pruned Minimax, the tree is pruned to only continue investigating good paths. The algorithm controls the range of values the tree should continue searching down, and we initialize this range to ± 1000 for generality.

4.0.2 Q Learning

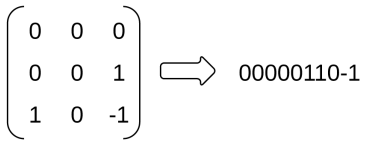
The first reinforcement learning approach we looked at was Q Learning, an off-policy TD control algorithm that we investigated in Practical 1. The straightforward version of Q Learning we implemented in Practical 1 did not seem like it could have any chance of performing as well as an AI algorithm such as Minimax, so we additionally implemented a number of extensions to Q Learning.

1. We replaced ϵ-greedy with the **VDBE-Softmax** policy described in [Tokic, 2011]. When evaluated on the Gridworld, VDBE-Softmax outperformed ϵ-greedy in cumulative reward substantially, so we thought it could potentially do better in Connect-N as well.

2. We implemented **eligibility traces**, as described in the Sutton and Barto textbook and as we covered in class discussions, as it can help the learner learn more efficiently. Unfortunately, the complete eligibility traces algorithm iterates over all states and actions, which was computationally infeasible in this game, as the states represent all possible versions of the Connect N board state. Instead, we implement a limited version of eligibility traces, where the algorithm updates $Q(s, a)$ and $e(s, a)$ for each board state it currently knows and all board states that can be reached by taking one action from the current board state.

3. We implemented a version of **known states and shaping rewards**, where some classes of board states are given a known value. Since our Q Learner only stores states as we reach them, due to space constraints, our implementation of known states does not initialize values for states. Instead, at each board state, we check for certain

conditions, and populate the value table then if the conditions are satisfied. First, for each possible action, we checked if the action would create a streak of length greater than or equal to $n-1$. If so, we give it a value of 15. For reference, a winning action has value 25. Second, we checked if the action would directly prevent an opponent win, and gave the action value 20, which guarantees our learner would prevent enemy wins. The implementation of known states for Q Learning was inspired and based on the heuristics for Minimax, since the idea of both is to estimate the value of a state.



The Q learner continues to keep an action value table (as in Practical 1), with a slightly different data structure. The Q learner takes the board state and hashes it into a string, where 0 represents an empty spot and 1 and -1 represent self and opponent tokens. It then adds the string board state as a key into a dictionary. For computational reasons, instead of adding all of the possible board states to the dictionary, the dictionary only ever holds board states that it has seen.

To pre-train the Q learner, we pass the board state between different games, allowing the Q learner to have a better, more competitive knowledge of what actions are good or bad.

Lastly, we explore training the Q learner over many many games, a strategy that we refer to as "pre-training." Since the Q learner initializes its action value table to zeros, when it pre trains for many games the Q learner begins a game with better knowledge of what board positions are potentially advantageous. Additionally, in our web interface, each game a human plays against the Q learner helps train its action-value table.

4.0.3 Monte Carlo Tree Search- andre

5 Results

Table 1: Player 1 Random

Player 2 Type	W/D/L
Random	0.5607 / 0.0002 / 0.4391
Extend	0.04 / 0.0 / 0.96
MCTS	0.2 / 0.0 / 0.8
Minimax d2	0.13 / 0.0 / 0.87
Minimax d2 ab	0.65 / 0.0 / 0.35
Minimax d4	0.01 / 0.0 / 0.99
Minimax d4 ab	0.02 / 0.0 / 0.98
Minimax d5	0.03 / 0.0 / 0.97
Minimax d5 ab	

Table 2: Player 1 Q-Learning Trained 10k

Player 2 Type	W/T/L
Random	0.7604 / 0.1118 / 0.1278
MCTS	0.75 / 0.05 / 0.2
Minimax d2	0.139 / 0.351 / 0.51
Minimax d4	0.0 / 0.4 / 0.6

Table 3: Player 1 Q-Learning Known States Trained 10k

Player 2 Type	W/D/L
Random	0.7502 / 0.1192 / 0.1306
Minimax d2	0.149 / 0.334 / 0.517
Minimax d4	0.0905 / 0.8438 / 0.0657

Table 4: Player 2 Minimax d2

Player 1 Type	W/D/L
Random	0.87 / 0.0 / 0.13
Extend	1.0 / 0.0 / 0.0
Minimax d2	1.0 / 0.0 / 0.0
Q Learning Trained 10k	0.51 / 0.351 / 0.139
Q Learning Known Trained 10k	0.517 / 0.334 / 0.149

Table 5: Player 2 Minimax d5

Player 1 Type	W/D/L
Random	0.99/ 0.0 / 0.01
Minimax d4	1.0 / 0.0 / 0.0
Minimax d4 (Forced Player 1 Move Middle)	0.0 / 0.0 / 1.0
Q Learning Trained 10k	0.6 / 0.4 / 0.0
Q Learning Known Trained 10k	0.512 / 0.487 / 0.001

5.0.4 Confirmations about general Connect-4 Strategies

In running our random learner against itself, we see that it is advantageous to go first, as the first random learner has a higher winrate. We theorize that this is because the first player potentially gets an extra turn, another token on the board to win with.

In running our Minimax learners against themselves, we notice that when Minimax 2 plays against Minimax 4, it is not guaranteed to win as it cannot "outthink" Minimax 4. Interestingly, Minimax 4 playing Minimax 5 often splits winrate and ties many times, indicating that at some point, iterating further down into the tree, while more computationally intensive, is not actually that important. This could be because at some depth, the agent has already learned what it needs to know to choose its immediate next action. Upon taking this next action, it rebuilds the tree anyway, meaning the additional depth exploration is actually not necessary.

Furthermore, when playing Minimax 4 against itself, we notice a few more results:

1. The second player wins if the first player does not go in the middle (odd number of columns)

2. The first player wins if it goes in the middle (odd number of columns). We execute this strategy by adding an optional parameter that forces Minimax to begin in the middle column.

We were surprised at first, but checking into the Connect-4 literature, we find that this is already a known fact. Given that an agent acts optimally, if it goes first and starts in the center row, it can always force a win. We discussed this in great depth, and believe that it allows the greatest flexibility in assuring diagonal wins that start the furthest down, which is important as executing a diagonal win strategy requires a large number of tokens to be put down. While horizontal wins are also require a large number of tokens, they are less common as more easily shut down and recognized.

5.0.5 Q Learning

The first thing we noticed was that the Q Learner did not appear to be learning much. After training our Q Learner for 10,000 games against many different strategies, the win rate was not much higher than using the untrained Q Learner. We theorize that this is because the Q learner's action-value table treats each board as a different state (including counting a slot filled with tokens by player 1 and player 2 as separate states), but in the test games that it plays, it sees very few games where there are exactly the same states as it knows. Thus, it is questionable whether or not eligibility traces and VDBE-Softmax helped the Q Learner's performance.

However, the Q Learner performed much better than we originally anticipated. We theorize this is a result of the different heuristics that we added. For example, if there are ties in the action table, instead of breaking ties randomly, we choose the first index when we exploit, which allows the Q Learner to "stack" its tokens all in one column. This often beats the random strategy easily, particularly if the random strategy does not "get lucky" and drop a token on top of the Q Learner's stack.

Additionally, we added a number of heuristics with our shaping rewards/known states. The shaping rewards/known states extension gives the Q Learner intermediate rewards for extending its own streaks and stopping opponent streaks, which allows the Q Learner to take more targeted actions that exploit more effectively. While the Q Learner does take exploratory actions, our version of Q Learning's exploit actions are much more strategized and targeted towards the Connect-4 game, making the exploitation much better than originally anticipated. This allows the Q Learner to perform in a way that often beats humans. Playing against our Q Learner online is far more difficult than originally anticipated.

Furthermore, we noticed that the Q Learner often loses against Minimax (as anticipated), but surprisingly is able to bring a large number of games to a tie. As we watched these games, we theorize that this is because the Q Learner can effectively "shut down" enough of the Minimax streaks that if Minimax does not build a situation where it can win in two (or more) ways, the Q Learner can fill up the board and force the game to a tie.

5.0.6 MCTS

We notice that MCTS does not perform as well as we expected. Based on the paper discussed in lecture, it seemed that MCTS would be able to drastically outperform Q Learning, Random, and Minimax 2, as it can asymptotically converge to Minimax. In our exploration of MCTS, however, we found the greatest gap between theory and practice. MCTS was computationally intensive and slow, as it required a large number of iterations to perform well. Additionally, we often found that MCTS continued simulating other possible games and moves when there was clearly a winning move to be taken, or a move it needed to take to prevent the opponent from winning.

We were not successful in addressing the prior issues of computational intensity. Our MCTS learner was only tested with 200 iterations, which we realized was too few for it to determine good, competitive moves, as the MCTS algorithm uses the iterations to explore deeper into each branch, rather than exploring multiple branches equally (like Minimax). It is possible that some branches that could potentially be valuable are not being explored.

We addressed the second issue using our extension that stops the MCTS iteration when it sees a winning move, which helped reduce computational time. However, this extension did not drastically change the winrate, leading us to theorize that asymptotically MCTS was taking the correct actions enough that we couldn't see the impact of potentially iterating beyond critical moves.

Another issue that could have contributed to subpar MCTS results was our default policy, which we set to be random. It seems that in playing Connect-4 well, heuristics matter a lot. For example, the Minimax algorithm before static evaluation and intermediate state values, and Q learning with shaping rewards that essentially tell the Q learner what it needs to do in the specific Connect-4 case, perform much worse. It seems as if we should add Minimax-like values and attach them to each MCTS board state, so that there is a better default policy of trying actions that lead to high-value board states. This could potentially also cut down on computational time, as MCTS would theoretically only explore actions that are potentially good. Intuitively, heuristics give the RL agent more information specific to its current task and provide some starting strategies on how to navigate and complete the task, which evidently give the agent a jump start over other agents that must learn information about the world and form strategies at the same time.

5.0.7 Extend

Our naive baseline of Extend represents how a human would play, trying to extend streaks as far as possible. Unlike a human, it does not try and prevent opponent streaks. Extend beats a random agent most of the time, even more than MCTS beats a random agent. We believe that this good performance of our Extend agent reveals that specific knowledge about the domain in the form of heuristics allow agents to perform quite well.

6 Future Work

Interesting research questions for the future could include:

1. Exploring extensions to MCTS, such as a better default policy (perhaps estimation of a value for each board state, and selecting the action that leads to the board state with highest value)

2. Exploring large N, where computational complexity of solving MCTS and Minimax could potentially allow Q learning to perform competitively (as MCTS iteration number would be low and Minimax depth parameter would be low)

3. Adding stochasticity to the board, such as dropping the bottom row with some probability or having a probability of taking the wrong action

4. Coding the board state as a POMDP, with information such as the number of tokens in a row, but not knowing exactly where the tokens are