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

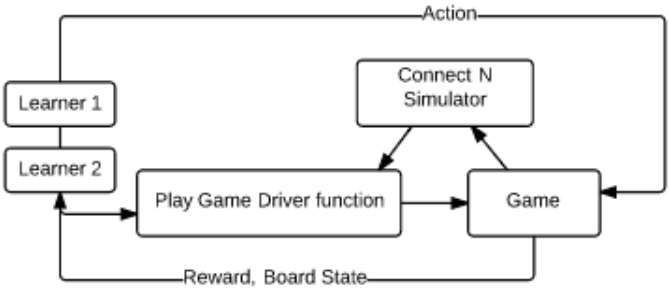


Figure 1: Play Game Driver Function

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:

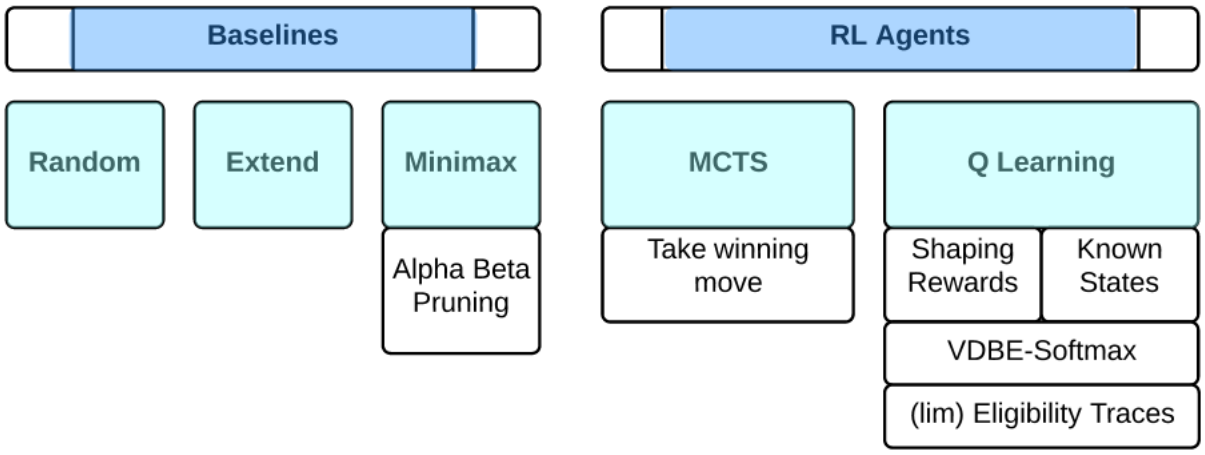


Figure 2: Implemented Baselines and Learners, with Extensions

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

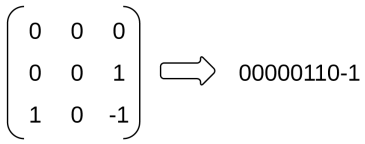
The first reinforcement learning approach we looked at was Q Learning, an off-policy TD control algorithm that we investigated in Practical 1. The straightforward version of Q Learning we implemented in Practical 1 did not seem like it could have any chance of performing as well as an AI algorithm such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

4.0.3 Monte Carlo Tree Search- andre

5 detailed description of your results/comparisons- everyone- wait for results

Player 1 (row) / Player 2 (col)	Random
Random	0.5607 / 0.0002 / 0.4391
Extend	
MCTS	
Minimax depth-2	
Minimax depth-2 'ab'	
Minimax depth-4	
Minimax depth-4 'ab'	
Minimax depth-5	
Minimax depth-5 'ab'	
Q-Learning	0.7604 / 0.1118 / 0.1278 (trained 10000) - 0.7493 / 0.1209 / 0.1298 (trained 22000)
Q-Learning w/ Known States	0.7502 / 0.1192 / 0.1306 (trained 10000) - 0.7503 / 0.1201 / 0.1296 (tra

6 thorough description of why you got the results you did, a few pages- everyone- wait for results

7 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