CONNECT FOUR

By: - Achal Vyas (UID: 116869560)

Github link: - https://github.com/Achalpvyas/Connect_4_using_RL

Introduction: -

Skillful game playing has always been considered a token of intelligence, consequently Artificial Intelligence and Machine Learning exploit games in order to exhibit intelligent performance. A game that has become a benchmark, exactly because it involves a great deal of complexity along with very simple playing rules, is the game of connect 4. My work focused on developing a program for a Connect Four game and have it train itself through self-play and reinforcement learning. Game consists of a grid board in which consist of a 6 x 7 tile layout. Each player may drop only one checker into the grid per turn.A winner is declared when one player gets four of their colored checkers in a row vertically, horizontally or diagonally. Other configurations, such as squares, don't count as winning combinations. The game is declared a stalemate or tie if all the pieces are played but nobody has four in a row.

It consist of multiple actions and even more number of possible states that this actions can lead to. These magnitudes are hard to deal with for any kind of player (human or computer). Adversarial Tetris is a variation of Tetris that introduces adversity in the game, making it even more demanding and intriguing; an unknown adversary tries to hinder the goals of the player by actively choosing pieces that augment the difficulty of line completion and by even "leaving out" a tile from the entire game, if that suits his adversarial goals. This report describes the design and implementation of an agent capable of learning to improve his strategy against any adversary. My program employs MiniMax search to produce a strategy that accounts for any adversary and reinforcement learning to learn an appropriate state evaluation function. Our agent exhibits improving performance over a wide range of adversaries and an increasing number of learning games.

Method: -

Minimax is a decision-making algorithm, typically used in a turn-based, two player games. The goal of the algorithm is to find the optimal next move.

In the algorithm, one player is called the maximizer, and the other player is a minimizer. If we assign an evaluation score to the game board, one player tries to choose a game state with the maximum score, while the other chooses a state with the minimum score.

In other words, the maximizer works to get the highest score, while the minimizer tries get the lowest score by trying to counter moves.

It is based on the zero-sum game concept. In a zero-sum game, the total utility score is divided among the players. An increase in one player's score results into the decrease in another player's score. So, the total score is always zero. For one player to win, the other one has to lose. Examples of such games are chess, poker, checkers, tic-tac-toe.

Connect 4 decides how to play by looking at the board positions that would result from each available move, evaluating each of them using a Neural Network, and then playing the best one. The technical term for the role that the Neural Network is playing in this system is a *Value Function Approximator* - when you show a board position to the network it gives you back a numeric score indicating how much value it assigns to that position

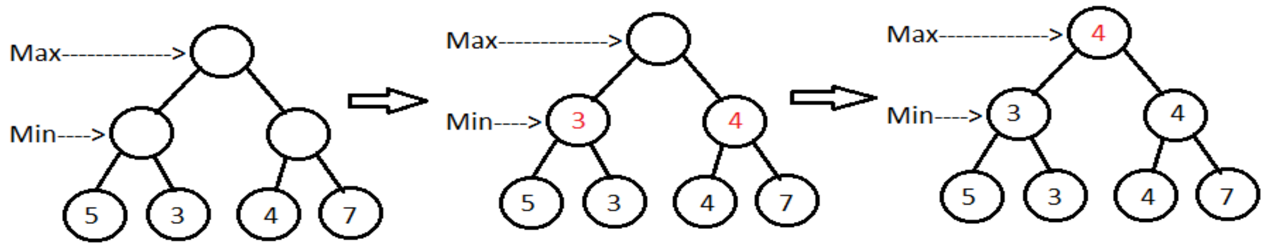
Approach: -

Our goal is to find the best move for the player. To do so, we can just choose the node with best evaluation score. To make the process smarter, we can also look ahead and evaluate potential opponent's moves.

For each move, we can look ahead as many moves as our computing power allows. The algorithm assumes that the opponent is playing optimally.

Technically, we start with the root node and choose the best possible node. We evaluate nodes based on their evaluation scores. In our case, evaluation function can assign scores to only result nodes (leaves). Therefore, we recursively reach leaves with scores and back propagate the scores.

Consider the below game tree:



Maximizer starts with the root node and chooses the move with the maximum score. Unfortunately, only leaves have evaluation scores with them, and hence the algorithm has to reach leaf nodes recursively. In the given game tree, currently it's the minimizer's turn to **choose a move from the leaf nodes**, so the nodes with minimum scores (here, node 3 and 4) will get selected. It keeps picking the best nodes similarly, till it reaches the root node.

Now, let's formally define steps of the algorithm:

1. Construct the complete game tree
2. Evaluate scores for leaves using the evaluation function
3. Back-up scores from leaves to root, considering the player type:
    - For max player, select the child with the maximum score
    - For min player, select the child with the minimum score
4. At the root node, choose the node with max value and perform the corresponding move.

Self-Play Reinforcement Learning:

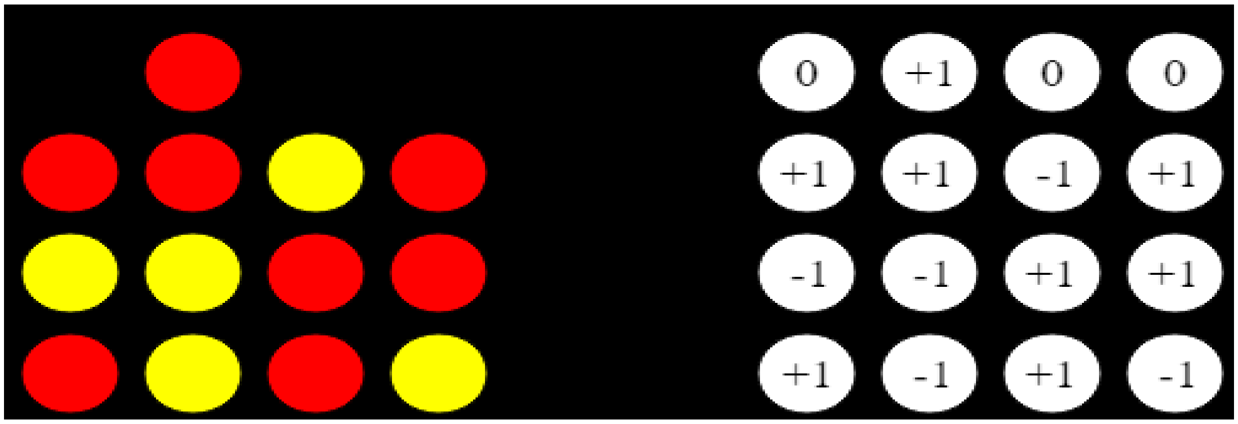
The Neural Network was trained using 'self-play', which is exactly what it sounds like: two opponents play many games against each other, both selecting their moves based on the scores returned by the network. As such, the network is learning to play the game completely from scratch with no outside help. The games that it plays against other types of opponent, such as MCTS, are used only to check its progress and the results of those games are not used for training.

At the end of each training game the network receives some feedback about how well its recommended moves worked. The scores for the board positions recommended to the losing player are all reduced slightly, and those recommended to the winning player are increased.

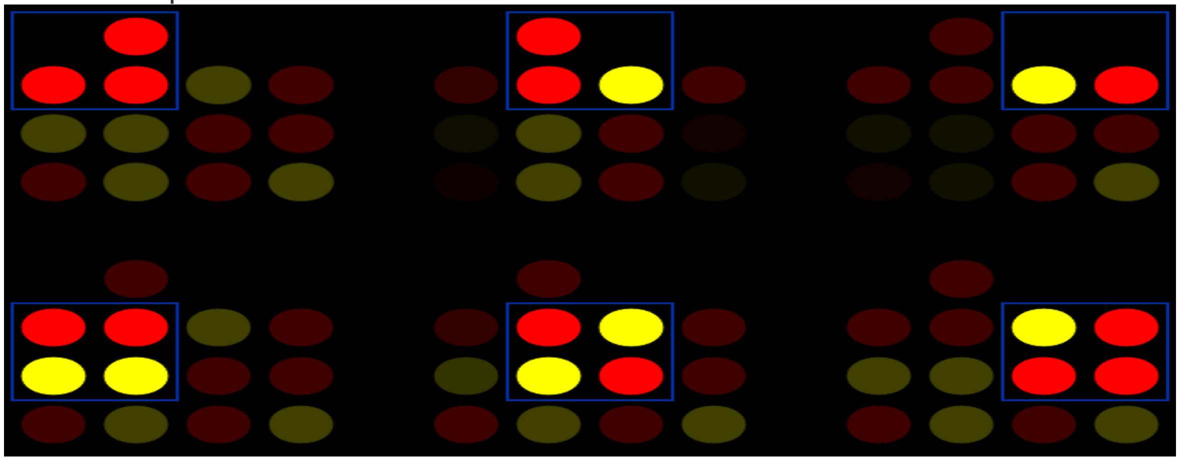
The scores for board positions occurring towards the end of the game (i.e. closer to the eventual victory or loss) are adjusted by larger amounts than the scores for earlier positions, in other words they are apportioned a larger amount of the credit or blame for what happened.

Convolutional Layers:

A Convolutional Layer takes a 2-dimensional grid of numbers as its input. In order to process a Connect 4 board, the board must be converted into a grid of numbers, for example by using '0' to represent an empty position, and '+1' and '-1' to represent positions occupied by the red and yellow pieces:



A convolutional layer processes its input using a 'filter' which can be thought of as a small square box. The filter is moved over the input from left to right and top to bottom covering the entire input field.



At each position the filter generates a set of numbers that in some way represent the input values at that position. The numbers produced by the filters at each position form a new grid of values, which then become the input for the next layer in the network.

A Fully Connected Layer comprises an array of 'neurons', each of which takes all the input values for that layer and produces a single output value. The combined output values of all the neurons in the layer become the input for the next layer in the network.

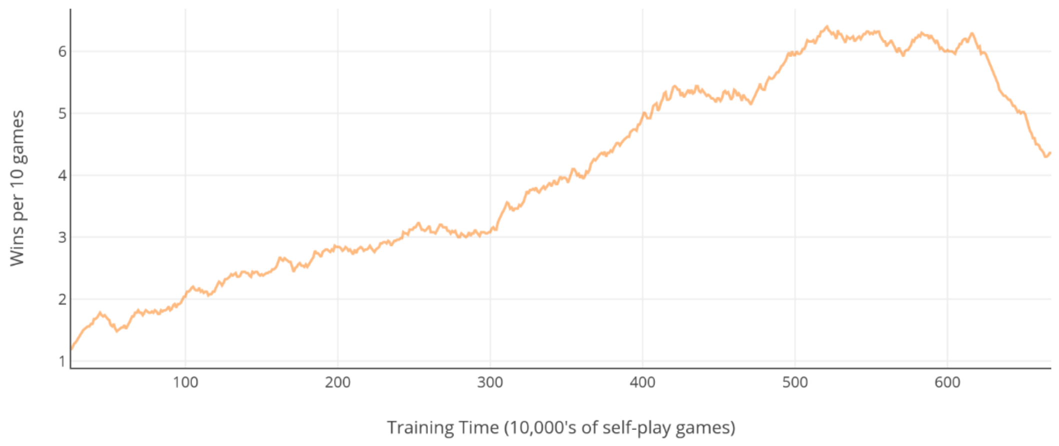
Moreover, I included the time delay to the AI piece so that a good user interface can be formed. For this I used {pygame.time.wait(500)}. Similar time delay is also implemented at the end of the game.

Also, I included the code will help to randomly initiate who goes first. For this I used {turn = random.randit(PLAYER, AI)}

Results: -

Minimax algorithm is one of the most popular algorithms for computer board games. It is widely applied in turn based games. It can be a good choice when players have complete information about the game.

Next, I ran a prolonged self-play training session, allowing the network to train for over 6 million games, testing it against a MCTS-1000 player every 10,000 games. The results were interesting - the network improved steadily for the first 5 million games, but then peaked at a level where it was winning around 60% of its games. After that point it actually started getting worse, and I stopped training once performance had dropped to around 40%.



Analysis and conclusion: -

I suspect that the drop in performance was caused by over-fitting. Over-fitting is a common problem in machine learning systems, it occurs when a model adapts to perform well on the data that is being used to train it, but fails to generalise and performs poorly on data that it was not exposed to during training. In this case I think the strategies which the network learned during the later part of its self-play training were giving good results in its training games, but were not effective when tested against an opponent that played in a different style.

For most of the problems, it is not feasible to construct an entire game tree. In practice, we can develop a partial tree (construct the tree till a predefined number of levels only). Then, we will have to implement an evaluation function, which should be able to decide how good the current state is, for the player. Even if we don't build complete game trees, it can be time-consuming to compute moves for games with high branching factor.

Future Work: -

Fortunately, there is an option to find the optimal move, without exploring every node of the game tree. We can skip some branches by following some rules, and it won't affect the final result. **This process is called pruning**. Alpha-beta pruning is a prevalent variant of minimax algorithm. It may not be the best choice for the games with exceptionally high branching factor. Thus, in future work I will aim to incorporate this pruning in order to improve the outputs.