Mawaba Pascal Dao
ML HW1
CSE5693
Dr. Chan
Feb 2, 2021

1. **Written part**
    a. **Exercise 1.2**

Informal task description:

The learning task we wish to learn is playing tic-tac-toe. Tic-tac-toe is a 2 player game with a 3x3 grid where players win by placing 3 of their symbols in a straight line; vertically, horizontally, or diagonally. The system will learn to intelligently play this game against an opponent, human or not.

Detailed description:

- **Task:** Playing tic-tac-toe
- **Performance measure:** Percent of games won against opponents
- **Training experience:** Playing practice games against itself or a human

Target function

The target function *V* is a function that takes in a board state and outputs a real number between -1 and 1 as the value of this board state such that:

- *V*(*b)* = 1 if *b* is a final state that is won
- *V*(*b)* = -1 if *b* is a final state that is lost
- *V*(*b)* = 0 if *b* is a final state that is won
- *V(b)* = *V(b')* where *b'* is the best final state achievable starting from b as described in sections 1.3 and 1.4 of the book.

Target function representation

The target function is represented as a linear combination of board state features. Board state features can be expressed using 2 different expressivity modes. The expressivity mode is defined by the user as an argument in the run command (view README.txt for more details). We define and implement the following expressivity modes:

- **Compact expressivity**
  The features used to represent the board state in this mode are:
    - x1: Number of X's on the board (System's marks)
    - x2: Number of O's on the board (Opponent's marks)
    - x3: Minimum number of moves for X to win
    - x4: Minimum number of moves for O to win

- **Full expressivity**
  The features used to represent the board state in this mode are:
    - x1: Number of X's on the board (System's marks)
    - x2: Number of O's on the board (Opponent's marks)
    - x3: Minimum number of moves for X to win in row 0
    - x4: Minimum number of moves for X to win in row 1
    - X5: Minimum number of moves for X to win row 2
    - x6: Minimum number of moves for O to win row 0
    - X7: Minimum number of moves for O to win row 1
    - X8: Minimum number of moves for O to win row 2
    - x9: Minimum number of moves for X to win in col 0
    - x10: Minimum number of moves for X to win in col 1
    - X11: Minimum number of moves for X to win col 2
    - x12: Minimum number of moves for O to win col 0
    - X13: Minimum number of moves for O to win col 1
    - X14: Minimum number of moves for O to win col 2
    - X15: Minimum number of moves for X to win diag 1
    - X16: Minimum number of moves for X to win diag 2
    - X17: Minimum number of moves for O to win diag 1
    - X18: Minimum number of moves for O to win diag 2

Compact expressivity yielded the best results during experiments so we will use compact expressivity for the rest of the discussion. Therefore V_hat is represented as:

$$\mathbf{V\_hat(b) = w0 + w1*x1 + w2*x2 + w3*x3 + w4*x4}$$

Tradeoffs
In formulating the compact expressivity it was important to carefully select board features that could distinctly represent winning, losing, or neutral states. The small number of features allows the system to train on a higher level representation thus requiring fewer training examples to learn the game. The main tradeoff with compact representations is limited expressivity. Because the compact representation is not expressive (only 4 features to represent the entire board) it can be harder for the system to correctly approximate *V.*

b. Exercise 1.4

- **Random legal board position**

This strategy provides good coverage of the space of possible board states. The program might become a good high-level beginner or intermediate player, but will likely not be good at certain niche states of interest. By randomly sampling the search space of board states, we cannot focus on specific regions such as situations where winning is not an option.

- **Picking up a board state from the previous game, then applying one of the moves that were not executed**

This strategy allows the program to explore different moves for the same board state and find the best move for a given board state. The downside is that it may limit the variety of board states the program is exposed to.

- **Custom strategy: Generate random board states 80% of the time. For the remaining 20% pickup a state from a previous lost game, and apply a move that was not executed.**

With this strategy, the program has a good general representation of how to play and win the game. But also learns-- through replaying lost games and finding strategies to avoid losing -- to avoid losing games where it can not win.

c. Learned weights

No teacher learned weights:

W = [1.1792, -0.3118, 0.2077, -0.3048, 0.0999]

We can see from these wights that w3 and w1 are negative and have the 2 highest absolute values, meaning they weigh more on the calculation of v_hat. These are the weights for a minimum number of moves for X to win, and the number of X's on the board respectively. As expected w3 is negative because having a higher number of moves to victory is a negative predictor of success, we prefer having a lower number of moves to victory. Perhaps more interestingly is w1–the weight for the number of X's– The naive assumption might be that we prefer having more X's on the board, but this is not what the system learns. This is possible because having multiple X's on the board is indicative of situations where victory was not obtained quickly, having fewer X's means winning faster.

Teacher learned weights:

W = [0.1093, -0.004, 0.0042, 0.0017, 0.0052]

Although w3 is not negative here, we observe that its value is smaller than w4. We also see that w1 is negative just as in the no-teacher case. It is also worth noting that these weights were obtained after training on only 246 games from the input file. In contract no-teacher trained on 30k games.