TIC TAC TOE AI MODEL

Using LMS error function

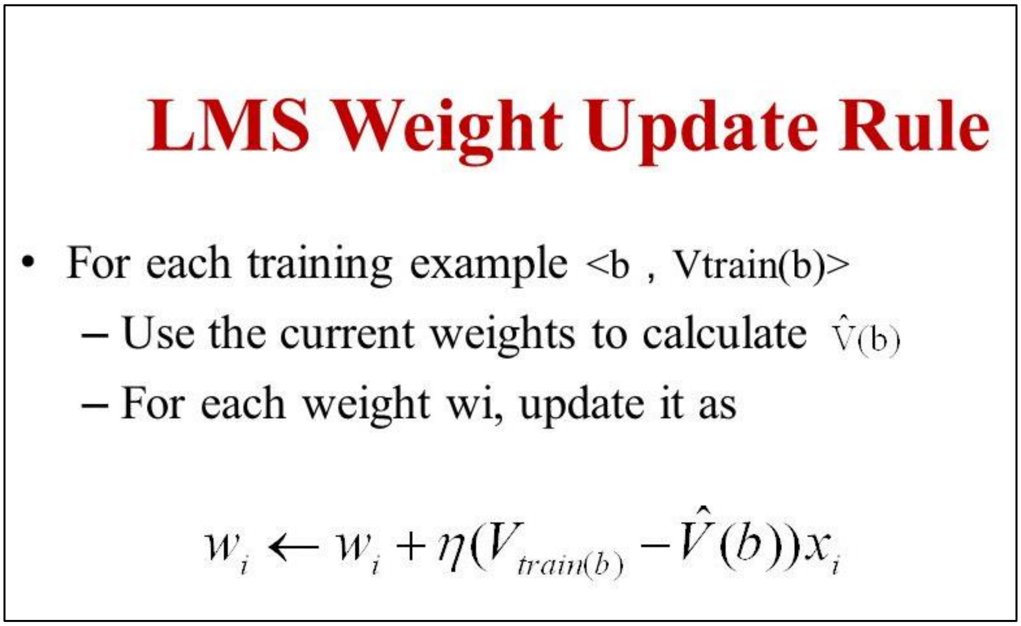
NAME: SRI SAI VIJAYA ADITYA NITTALA

ROLL NO.: 177163

Section: A

The **Least Mean Squared** weight update rule is derived from the **Least Mean Squared** error function. This function is essentially average of the square of the distance between our prediction and the target value.

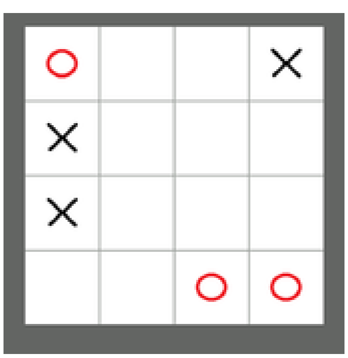
The weight update rule is:



Where:

- *Vtrain(b)* is the target for the training example *b*.
- *V(b)* is the hypothesis obtained from the weighted sum of input features and weights.
- η is the learning rate.

Input feature extraction from a given 4x4 Tic Tac Toe board:



There are **8*size + 1** number of features:

- 1 is for the bias
- For n {from 1 to size}:
  - Number of rows with n Xs
  - Number of rows with n Os
  - Number of columns with n Xs
  - Number of columns with x Os
  - If major diagonal has n Xs
  - If major diagonal has n Os
  - If minor diagonal has n Xs
  - If minor diagonal has n Os

Assignment - 4 : Tic Tac Toe AI using LMS rule

NAME: SRI SAI VIJAYA ADITYA NITTALA

ROLLNO: 177163

SECTION: A

NOTE:

> The following program implements Tic Tac Toe for a 4x4 board due to insufficient infrastructure, that is, takes too long to play for a 5x5 board.

```
import numpy as np
import pandas as pd
import copy
import math
from tqdm import tqdm


def initialize_board(size = 4):
    '''
    Initialize board: (4x4)
    --------------------
    | - || - || - || - |
    --------------------
    | - || - || - || - |
    --------------------
    | - || - || - || - |
    --------------------
    | - || - || - || - |
    --------------------
    '''
    board = np.ones((size, size), dtype = str)
    board[:,:] = '-'
    return board


def getWinner(board):
    '''
        Function to return the winner based on the given board
        +100 : for X
        -100 : for O
    '''
    size = board.shape[0]

    for i in range(size):
        if np.all(board[i,:] == 'X'): # 4 Xs in row i -> X
            return 100
```

```
        if np.all(board[i,:] == '0'): # 4 Os in row i -> O
            return -100
        if np.all(board[:,i] == 'X'): # 4 Xs in col i -> X
            return 100
        if np.all(board[:,i] == 'O'): # 4 Os in col i -> O
            return -100

    diagonally = [1, 1, 1, 1]

    for i in range(size):
        if board[i,i] != 'X':
            diagonally[0] = 0
        if board[i,i] != 'O':
            diagonally[1] = 0
        if board[i,(-i-1)] !=  'X':
            diagonally[2] = 0
        if board[i,(-i-1)] !=  'O':
            diagonally[3] = 0

    for i in range(4):
        if diagonally[i]:
            if i == 0 or i == 1:
                return 100
            else:
                return -100


    ### check if it was a draw
    if np.sum(board == '-') == 0:
        return 0

    # Otherwise not over
    return -1


def ExctractFeatures(board):
    '''
        Function to extract the (size*8 + 1) features from the board
        Explanation of input features given in assignment report
    '''

    # get size and create feature vector using it
    size = board.shape[0]
    feature_vector = np.zeros(size*8 + 1, dtype=np.float64)

    # bias
    feature_vector[0] = 1.0

    # calculate the number of Xs, Os, and -s in rows and columns
    num_x_row   = np.count_nonzero(board == 'X', axis = 1)
    num_x_col   = np.count_nonzero(board == 'X', axis = 0)
    num_o_row   = np.count_nonzero(board == 'O', axis = 1)
```

```
    num_o_col   = np.count_nonzero(board == 'O', axis = 0)
    num_emp_row = np.count_nonzero(board == '-', axis = 1)
    num_emp_col = np.count_nonzero(board == '-', axis = 0)

    # diagonal counters
    major_X = 0      # Xs in leading
    major_O = 0      # Os in leading
    major_emp = 0    # -s in leading
    minor_X = 0     # Xs in trailing
    minor_O = 0     # Os in trailing
    minor_emp = 0   # -s in trailing

    for i in range(size):
        if board[i,i] == 'X':
            major_X += 1
        if board[i,i] == 'O':
            major_O += 1
        if board[i,i] == '-':
            major_emp += 1
        if board[i,-i-1] == 'X':
            minor_X += 1
        if board[i,-i-1] == 'O':
            minor_O += 1
        if board[i,-i-1] == '-':
            minor_emp += 1


    # add values to feature vector
    for i in range(1,size+1): # for each n of {1 to size}
        # for each row/col -> in first 4 of 8 positions
        for j in range(size):

            if num_x_row[j] == i and num_emp_row[j] == size-i:
                feature_vector[(i-1)*8 + 1] += 1
            if num_x_col[j] == i and num_emp_col[j] == size-i:
                feature_vector[(i-1)*8 + 2] += 1
            if num_o_row[j] == i and num_emp_row[j] == size-i:
                feature_vector[(i-1)*8 + 3] += 1
            if num_o_col[j] == i and num_emp_col[j] == size-i:
                feature_vector[(i-1)*8 + 4] += 1

        # diagonals -> in next 4 positions
        if major_X == i and major_emp == size-i :
            feature_vector[(i-1)*8 + 5]+=1
        if minor_X == i and minor_emp == size-i :
            feature_vector[(i-1)*8 + 6]+=1
        if major_O == i and major_emp == size-i :
            feature_vector[(i-1)*8 + 7]+=1
        if minor_O == i and minor_emp == size-i :
            feature_vector[(i-1)*8 + 8]+=1

    return feature_vector
```

```
    return feature_vector
```

Determine next possible moves

```
def getPossibleStates(board, player):
    '''
        Function to determine the next possible moves from the current board state.
        For every empty slot '-':
          Replace with either 'X' or 'O' and append it to a list
          return list
    '''
    board_states = []
    size = board.shape[0]

    for i in range(size):
        for j in range(size):
            if board[i,j] == '-':
                temp_board = copy.deepcopy(board)
                temp_board[i,j] = player
                board_states.append(temp_board)

    return board_states
```

Get the value of the given board

```
def XFWT_single(board):
    '''
      X : Extract
      F : Features
      W : Weighted
      T : Total

      For a given board, extract the features and then
      return weighted sum of features.
    '''
    # extract features and calculate value
    feature_vector = ExctractFeatures(board)
    board_value = np.dot(weights,feature_vector.T)

    return board_value


def XFWT(boards):
    '''
      X : Extract
      F : Features
      W : Weighted
      T : Total
    '''
```

```
    board_values = []

    for board in boards:
        # extract features and calculate value
        feature_vector = ExctractFeatures(board)
        board_value = np.dot(weights,feature_vector.T)
        board_values.append(board_value)

    return board_values
```

Printing Board

```
def printBoard(board):
    size = board.shape[0]

    print("-----"*size)
    for i in range(size):
        for j in range(size):
            print("| {} |".format(board[i][j]), end='')
        print()
        print("-----"*size)
```

Training function

```
def train(size, alpha, num_iters, weights):
    for epoch in tqdm(range(num_iters)):

        curr_board_history = []
        board = initialize_board(size)
        current = 'X'

        # till not over
        while (getWinner(board) == -1):

            # get next possible states, and calculate their values
            next_states = np.array(getPossibleStates(board, current))
            np.random.shuffle(next_states)
            next_values = XFWT(next_states)

            # append board to history and set the next board state as the one with maximum va
            curr_board_history.append(board)
            board = next_states[np.argmax(next_values)]

            # randomly change player to get optimal training
            rand_val = np.random.randn(1)
            if (rand_val > 0):
                current = 'X' if (current == 'O') else 'O'
```

```
        curr_board_history.append(board)
        result = getWinner(board)

        # update weights
        for idx, board_state in enumerate(curr_board_history):
            X = ExctractFeatures(board_state)
            if ((idx+2) < len(curr_board_history)):
                weights += alpha * (XFWT_single(curr_board_history[idx+2])
                              - XFWT_single(board_state))* X
            else:
                weights += alpha * (result - XFWT_single(board_state)) * X



# TRAINING
size = 4
alpha = 0.05
num_iters = 1000
num_features = size*8 + 1

weights = np.random.randn(num_features)*0.01 # initialize weights
train(size, alpha, num_iters, weights) # train model
```

```
100%|██████████| 1000/1000 [00:22<00:00, 45.27it/s]
```

Test Function

```
def play(size):
    board = initialize_board(size)
    current = 'X'

    # till game is not over
    while (getWinner(board) == -1):
        if current == 'X':
          print('Computer\'s turn : ')

          next_states = np.array(getPossibleStates(board, current))
          next_values = XFWT(next_states)
          board = next_states[np.argmax(next_values)]
          printBoard(board)
        else:
          print('Human\'s move : ')
          while (True):
            a = int(input('Enter x coordinate : '))
            b = int(input('Enter y coordinate : '))
            if board[a,b] == '-':
              break
            else:
              print("Please select an open position. Try again.")
```

```
          board[a,b] = 'O'
          printBoard(board)

        current = 'X' if (current == 'O') else 'O'

    result = getWinner(board)
    if result == 100:
      print('Computer Wins!')
    elif result == -100:
      print('Human Wins!')
    else:
      print('Draw!')


print("Let the game begin : ")
play(size)
```

```
| X || - || O || O |
--------------------
| - || - || - || X |
--------------------
Computer's turn :
--------------------
| X || X || - || O |
--------------------
| O || - || X || O |
--------------------
| X || - || O || O |
--------------------
| X || - || - || X |
--------------------
Human's move :
Enter x coordinate : 0
Enter y coordinate : 2
--------------------
| X || X || O || O |
--------------------
| O || - || X || O |
--------------------
| X || - || O || O |
--------------------
| X || - || - || X |
--------------------
Computer's turn :
--------------------
| X || X || O || O |
--------------------
| O || - || X || O |
--------------------
| X || - || O || O |
--------------------
| X || - || X || X |
--------------------
Human's move :
Enter x coordinate : 1
Enter y coordinate : 1
```

```
--------------------
| X || X || O || O |
--------------------
| O || O || X || O |
--------------------
| X || - || O || O |

--------------------
| X || - || X || X |
--------------------
Computer's turn :
--------------------
| X || X || O || O |
--------------------
| O || O || X || O |
--------------------
| X || - || O || O |
--------------------
| X || X || X || X |
--------------------
Computer Wins!
```