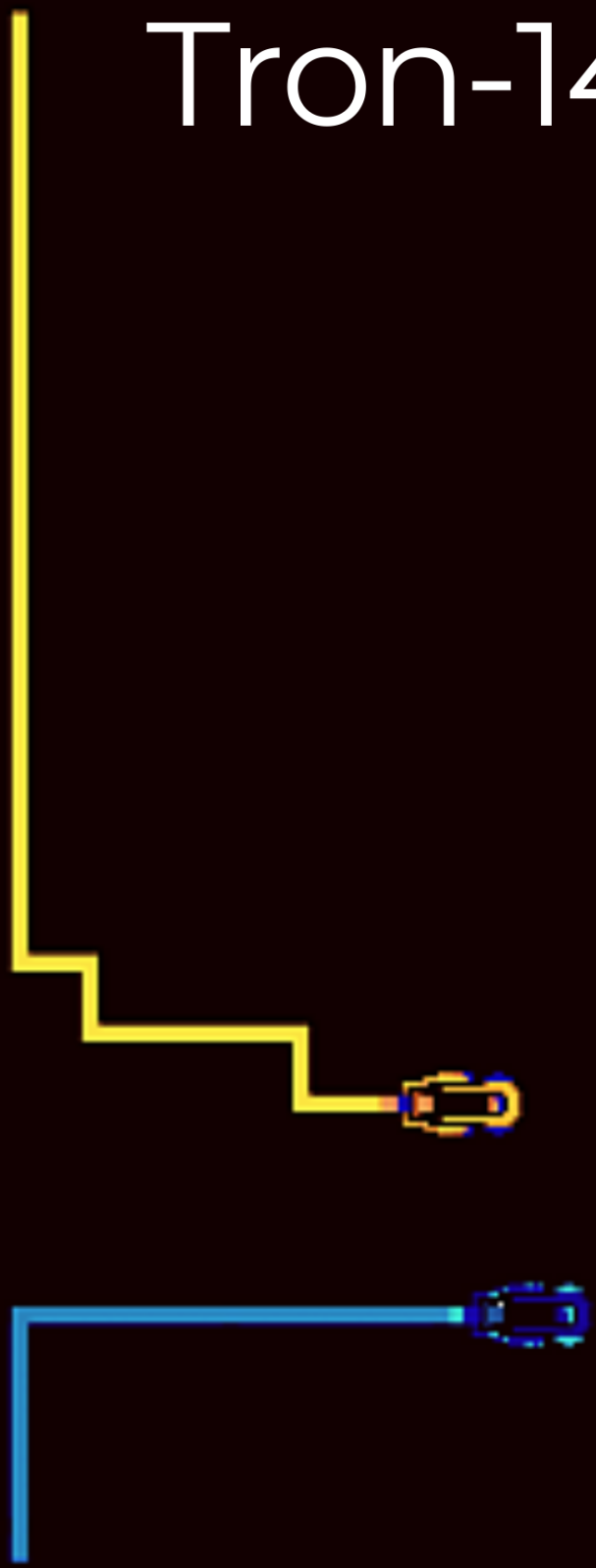
# Tron-141



DECEMBER 2020

CSCI 1410 | ARTIFICIAL INTELLIGENCE

# OVERVIEW

Welcome to my Tron robot, CountBOTula! I will discuss the background and detail the implementation of my CountBOTula throughout this writeup as well as outline several experiments I ran to improve the performance of my algorithm.

# BACKGROUND

I started out with a basic bot that employed adversarial search to win against RandBot consistently 80+% of the time, against WallBot 60+% of the time, TA1 bot more than 40% of the time, and TA2 bot only a few times. More specifically, my initial implementation used alpha-beta cutoff to choose the best action. In addition to the main *decide()* function, I included two helper functions, *get_ab_cutoff_min()* and *get_ab_cutoff_max()* that calculated the minimum value and the maximum value of the corresponding player's turn, respectively. I decided to use a very simple heuristic function for this initial warmup bot in order to have a baseline adversarial search algorithm to work with as I implemented my final bot. Therefore, for my heuristic, I simply calculated the number of free spaces on the board to determine which spaces the player could move to. This took into account spaces that did not include any walls, barriers, or another player. Overall, this approach worked but could certainly be improved on. This led me to developing an adversarial search algorithm with a different heuristic: one that

utilizes Voronoi regions, Dijkstra's search, and supervised learning to return a more effective heuristic value, leading to more wins in the game.

Before deciding on adversarial search, I also tried using a reinforcement learning strategy in which CountBOTula received a higher reward if it took an action that resulted in a win and was penalized if it took an action resulting in a loss. Using a basic reinforcement learning algorithm similar to the RL assignment, my bot did not perform very well against TA1 bot and TA2 bot.

```
self.alpha = learning_rate
self.epsilon = epsilon
self.gamma = gamma
self.lambda_value = lambda_value
rewards_each_learning_episode = []
for i in range(num_episodes):
    state = env.reset()
    action = self.LearningPolicy(state)
    episodic_reward = 0
    self.etable = np.zeros((self.num_states, self.num_actions))
    while True:
        next_state, reward, done, info = env.step(action)

        action2 = self.LearningPolicy(next_state)

        error = reward + self.gamma + self.qtable[next_state][action2] - self.qtable[state][action]

        self.etable[state][action] = 1

        self.qtable[state][action] += self.alpha * error * self.etable[state][action]

        self.etable[state][action] *= self.gamma * self.lambda_value

        state = next_state
        action = action2

        episodic_reward += reward

        if done:
            break

    rewards_each_learning_episode.append(episodic_reward)
```

*Figure 1: A reinforcement learning approach using the SARSA-λ algorithm to assign rewards to certain actions stored in the qtable and episodic rewards stored in the etable*

I decided to take a supervised learning approach because with the time constraint, it seemed more straightforward - however, I would potentially revisit reinforcement learning and Monte Carlo Tree Search if I had more time in order to see how it compares with a supervised adversarial search approach. Additionally, I chose adversarial search since solely the heuristic could be changed if I wanted to optimize the algorithm, which is exactly what I planned to do.

I mapped out my entire plan for an alpha beta cutoff algorithm using a supervised learning heuristic, as outlined in Implementation, before getting started on the code.

# IMPLEMENTATION

## *PERFORMANCE*

CountBOTula incorporates several different approaches to reach an effective solution that wins against RandBot 95+% of the time, against WallBot also 95+% of the time, against TA1Bot 90+%, and against TA2Bot 85+% of the time in the empty room:

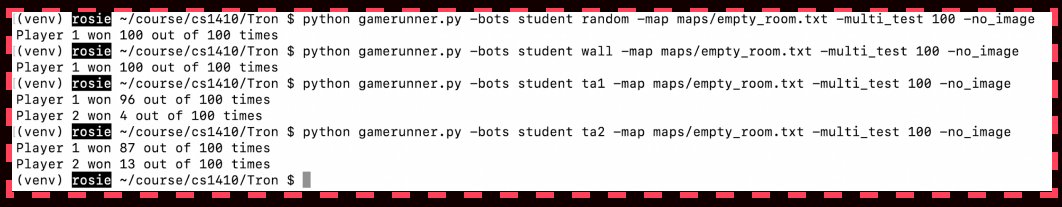



*Figure 2: CountBOTula (Player 1) performance against RandBot, WallBot, TA1, and TA2, respectively on the empty_room.txt map*

In the *center_block.txt* map, CountBOTula wins against RandBot and WallBot also 95+% of the time, against TA1Bot 80+% of the time, and against TA2Bot 75+% of the time (each TA1Bot and TA2Bot most of the time):

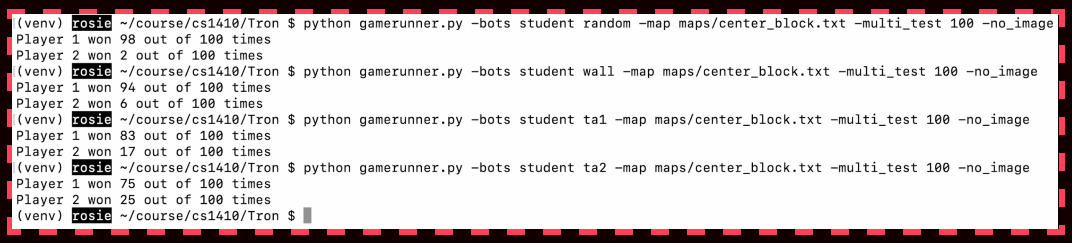



*Figure 3: CountBOTula (Player 1) performance against RandBot, WallBot, TA1, and TA2, respectively on the center_block.txt map*

Finally in the *diagonal_blocks.txt* map, CountBOTula wins against RandBot and WallBot 95+% of the time, against TA1Bot 80+% of the time, and against TA2Bot 75+% of the time (against TA1Bot and TA2Bot most of the time):

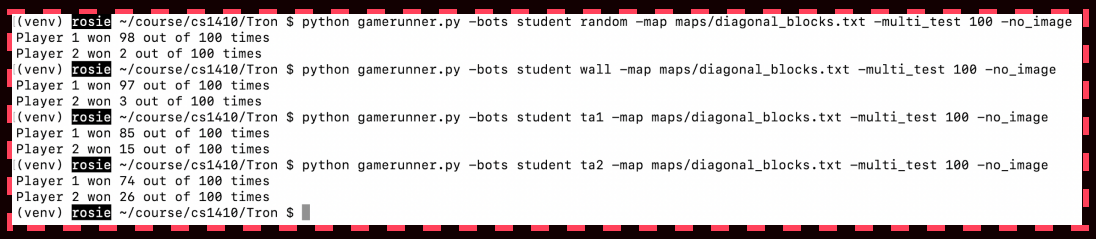



*Figure 4: CountBOTula (Player 1) performance against RandBot, WallBot, TA1, and TA2, respectively on the diagonal_blocks.txt map*

I also tested CountBOTula on maps of greater than 13 rows x 13 columns, and it wins against the four given bots most of the time.

## *THE VORONOI + DIJKSTRA HEURISTIC*

Initially, I used a plain Voronoi heuristic with my alpha beta cutoff algorithm. This approach labeled each space on the passed-in state's board as closer to Player 1 or Player 2, measured by the Manhattan distance, the sum of the absolute value of the differences between corresponding row and column locations between each player and a space. Player 1's Voronoi region is labeled *p1_region* in the code below, and Player 2's Voronoi region is labeled *p2_region*. The difference between the sizes of these regions was my initial heuristic in which the player whose Voronoi region was larger was the winner. However, this did not perform very well against any of the bots since it only took into account

perimeter barriers and not barriers that are located in the center of the map or in the player's path:

```
def plain_voronoi(self, state):
    """
    Evaluates Tron heuristic using plain Voronoi approach

    Input: state
    Output: heuristic value (int, float)
    """

    p1_region = []
    p2_region = []


    for row_index in range(len(state.board)):
        for col_index in range(len(state.board[row_index])):
            if state.board[row_index][col_index] == ' ':
                manhattan_from1 = abs(row_index - state.player_locs[0][0]) + abs(col_index - state.player_locs[0][1])
                manhattan_from2 = abs(row_index - state.player_locs[1][0]) + abs(col_index - state.player_locs[1][1])
                if manhattan_from1 > manhattan_from2:
                    p1_region.append(state.board[row_index][col_index])
                elif manhattan_from2 > manhattan_from1:
                    p2_region.append(state.board[row_index][col_index])
                else:
                    p1_region.append(state.board[row_index][col_index])
                    p2_region.append(state.board[row_index][col_index])

    p1_size = len(p1_region)
    p2_size = len(p2_region)


    difference = p1_size - p2_size

    return difference
```

*Figure 5: My initial Voronoi heuristic for adversarial search*

Therefore, I knew I had to incorporate some method that would take into account barriers and more appropriately calculate the shortest path distance from a given space to a player. I began brainstorming graph traversal methods and settled on Dijkstra's algorithm, transforming the board into a graph that the algorithm traverses and finds the shortest paths between spaces represented by graph nodes. I used Dijkstra's algorithm for other computer science classes and decided to see how it performed in conjunction with the Voronoi method, in the game of Tron. It ended up performing efficiently, resulting in the winning percentages shown in Figures 2-4.

```python
def eval_func(self, state):
    """
    Evaluates Tron heuristic using plain Voronoi with Dijkstra approach

    Input: state
    Output: heuristic value (int, float)
    """

    #Get player locations as tuples:
    p1_loc = state.player_locs[0]
    p2_loc = state.player_locs[1]

    #Get list of costs for each player by calling Dijkstra algorithm:
    p1_costs = self.get_costs(state, p1_loc)
    p2_costs = self.get_costs(state, p2_loc)

    p1_count = 0
    p2_count = 0

    #Player cost cannot go beyond this max cost of the board:
    board_cost = len(state.board) + len(state.board)

    #Loop through board and compute which player cost value is lower, add to player count if so:
    for row_index in range(len(state.board)):
        for col_index in range(len(state.board)):
            if p1_costs[row_index, col_index] < p2_costs[row_index, col_index] and p1_costs[row_index, col_index] <= board_cost:
                p1_count += 1
            if p2_costs[row_index, col_index] < p1_costs[row_index, col_index] and p2_costs[row_index, col_index] <= board_cost:
                p2_count += 1

    #Compute and return the difference in the counts (Voronoi method):
    difference = (p1_count - p2_count) / float(len(state.board) * len(state.board))

    return difference
```

*Figure 6: The evaluation function that incorporates Dijkstra's algorithm, self.get_costs(), to ultimately determine which player's Voronoi region is larger*

```python
def get_costs(self, state, loc):
    """
    Helper function that returns list of distances computed by Dijkstra

    Input: state, loc = location of passed in player
    Output: list of costs (distances)
    """

    #Get row and column of a single player's location:
    p_row = loc[0]
    p_col = loc[1]

    #Initialize costs and visited arrays which will keep track of costs and visited spaces, respectively:
    costs = np.zeros((len(state.board), len(state.board)))
    costs[:] = math.inf
    costs[p_row, p_col] = 0.0
    visited_spaces = np.zeros((len(state.board), len(state.board)))

    #Call get_neighboring_cells() helper function to find coordinates of spaces neighboring (p_row, p_col):
    neighbors = self.get_neighboring_cells(state, (p_row, p_col))

    #Loop through neighbors and populate costs array with cost of 1 since each neighbor is by definition one space away from cell:
    for neighbor_loc in neighbors:
        row, col = neighbor_loc
        costs[row, col] = 1

    #Initialize deque which I use to keep track of whether space is met:
    Q = deque(self.get_neighboring_cells(state, (p_row, p_col)))

    #As long the deque is not empty, go through all neighboring cells and update costs and visited arrays:
    while len(Q) != 0:
        #Use popleft() instead of pop() since it is optimized for a deque:
        curr_row, curr_col = Q.popleft()
        neighbor_cost = costs[curr_row, curr_col] + 1
        for neighbor_location in self.get_neighboring_cells(state, (curr_row, curr_col)):
            neighbor_row, neighbor_col = neighbor_location
            if neighbor_cost < costs[neighbor_row, neighbor_col]:
                costs[neighbor_row, neighbor_col] = neighbor_cost
            if visited_spaces[neighbor_row, neighbor_col] == 0:
                Q.append(neighbor_location)
                visited_spaces[neighbor_row, neighbor_col] = 1

    #Return the costs list, which will be used in the main eval_func() and multiplied by my supervised learning weights:
    return costs
```

*Figure 7: My Dijkstra implementation in which I update each player's cost list as well as visited spaces*

## *SUPERVISED LEARNING*

In order to incorporate a machine learning component in my algorithm, I decided to take a supervised learning approach from modifying *gamerunner.py* to generate a dataset to train my model in *tronmodel.py*, incorporating it into my adversarial search heuristic in *bots.py*, and testing it on unseen games between CountBOTula and the provided bots. My dataset, *tron_pickle15* is comprised of board state configurations and corresponding winners of Tron matches between TA2Bot vs. TA2Bot, TA1Bot vs. TA2Bot, TA1Bot vs. WallBot, and TA2Bot vs. WallBot. I did not include any data from RandBot or my own CountBOTula in the training set. Initially, I only included data from games with TA2Bot vs. TA2Bot but found that incorporating a more diverse set of examples would improve my model's accuracy.

In *gamerunner.py*, I store each game's list of boards (one board state per player move) as a Python list of lists in which each sublist in the board's list is a single row on the board. I pickle this file and then load it in *tronmodel.py* for preprocessing. To represent my data in such a way that barriers and visited spaces would be marked as "un-visitable" spaces and each of the player's locations could be represented, I chose to represent each example as a tuple of a concatenated vector consisting of a total of 338 entries (169 + 169 entries, each 13 x 13 representing a single state's board) where the first 169 entries represent a single board in which 1's represent occupied spaces and 0's represent free spaces. The second set of

169 entries are all 0's, except the locations of Players 1 and 2, where 1 represents Player 1 and -1 represents Player 2, respectively. Finally, the second entry of the tuple is binary and represents the example's label: a win (1) or a loss (0). Therefore, boards from the same game and boards of games with the same outcome will have the same label.


*Figure 8: A snapshot of a single example in my dataset, tron_training, where the first 169 spots represent the entries converted into 1's and 0's. 1's represent occupied spaces, including those marked by barriers ('#'), visited ('x'), or players ('1' or '2'), and 0's represent free spaces. The second set of 169 spots represents the entries of the board converted into 0's, a 1, and a -1, where 1 represents the location of Player 1, -1 represents the location of Player 2, and 0's represent all other spaces (this is a one-hot encoded vector). The corresponding label is either 0 or 1 in which 0 represents a loss and 1 represents a win.*

After preprocessing the data, I created a regression supervised learning model, including *train()* and *predict()* functions. This model predicts the probability that a player will win a game based on a given board state. I chose to use a regression model over a classification model since it is ok to classify an action as a winning or losing action, but it is even more helpful to distinguish between winning actions and determine which intermediate actions, with high or low probability (for example, 80% chance of winning versus a 30% chance of winning) result in a win.

Initially, my model did not perform very well so I decided to combine it with my Voronoi-Dijkstra approach explained above. I

take a weighted sum of the weights produced by my supervised learning model within my Voronoi heuristic.

```
[ 5.15535393e-04  5.15535393e-04  5.15535393e-04  5.15535393e-04
  5.15535393e-04  5.15535393e-04  5.15535393e-04  5.15535393e-04
  5.15535393e-04  5.15535393e-04  5.15535393e-04  5.15535393e-04
  5.15535393e-04  5.15535393e-04 -1.30928924e-02 -1.74985692e-02
  3.60150538e-02  1.21792882e-02 -4.95840874e-02  6.09615833e-03
  2.80576256e-03  3.19103466e-03  1.16994964e-02  1.36318952e-02
  1.49396293e-02  5.15535393e-04  5.15535393e-04 -1.02188467e-02
 -1.38754057e-02  4.29520127e-02  3.04200203e-02 -1.40995760e-02
  9.51405899e-03  6.49604159e-03  2.15551985e-03  5.81570849e-03
  9.15092800e-04  1.51779521e-02  5.15535393e-04  5.15535393e-04
 -7.37377113e-03 -1.04382936e-02 -2.94834280e-02 -1.41419030e-02
 -6.87918971e-03 -2.59443234e-03 -6.68923716e-03  2.40419381e-03
 -2.29778309e-03  6.26604327e-03  3.14773863e-02  5.15535393e-04
  5.15535393e-04 -5.36757547e-03 -9.68581003e-03 -8.96494257e-03
 -1.37134203e-02 -8.66889732e-03 -2.23506862e-03 -1.45376659e-02
 -1.09230656e-02  8.48588502e-03  8.45938331e-03  2.51300096e-02
  5.15535393e-04  5.15535393e-04  4.48918840e-04 -1.70727903e-03
 -8.50561700e-03 -5.74384627e-03 -1.13484784e-03 -2.38387010e-03
  4.48683816e-04  7.07051595e-03  3.51260753e-03  1.13855909e-02
```

*Figure 9: A sample of my model weights after training*

# EXPERIMENTS

The cutoff used in my adversarial search algorithm is not considered a hyperparameter, but I included it here to show how changing the hyperparameters, learning rate, $\alpha$, and the number of epochs alters my regression model's performance. My alpha-beta cutoff search algorithm uses a cutoff of 4.

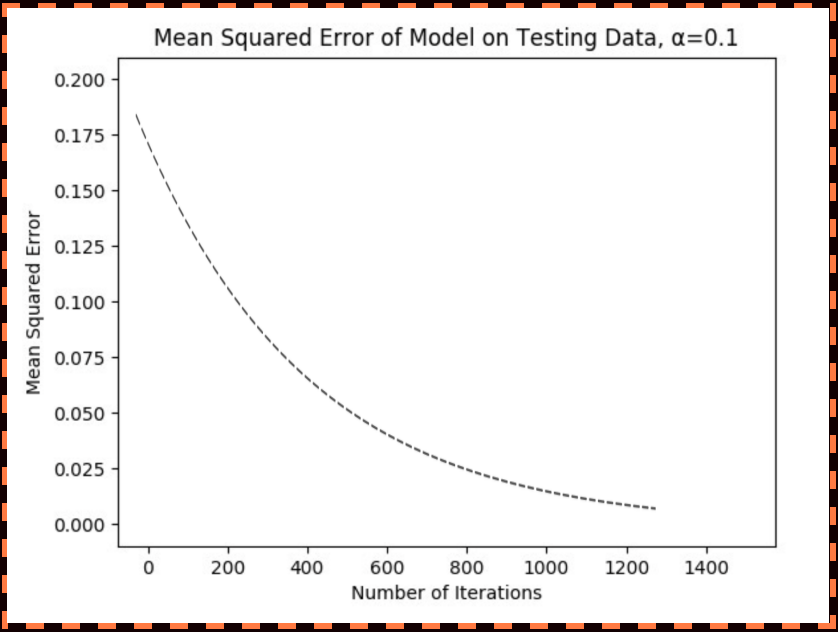



*Figure 10: A plot of model Mean Squared Error (MSE) as the number of iterations increases, $\alpha$ = 0.1, number of epochs = 1500, cutoff = 4*

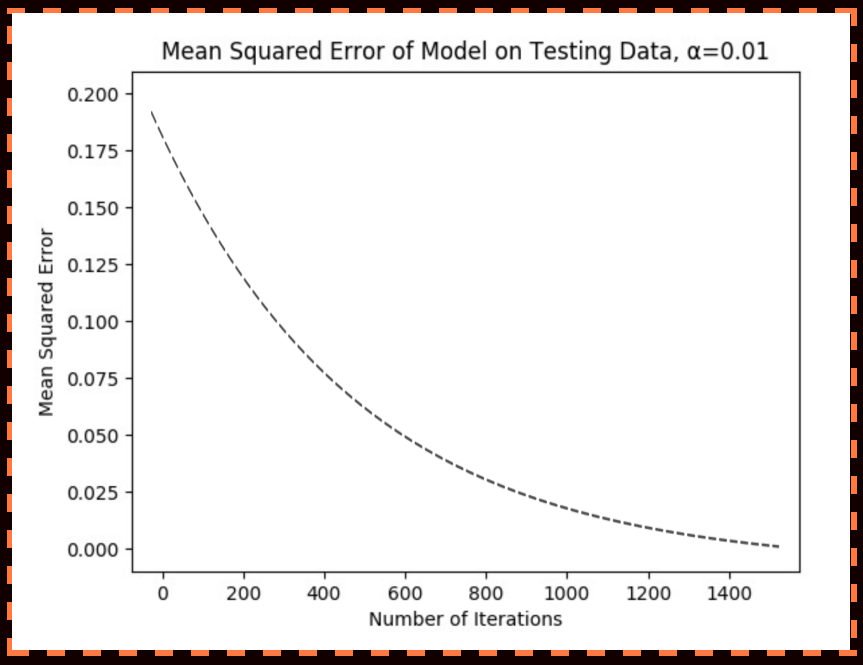



*Figure 11: A plot of model Mean Squared Error (MSE) as the number of iterations increases, α = 0.01, number of epochs = 1500, cutoff = 4 (optimal hyperparameter optimization)*

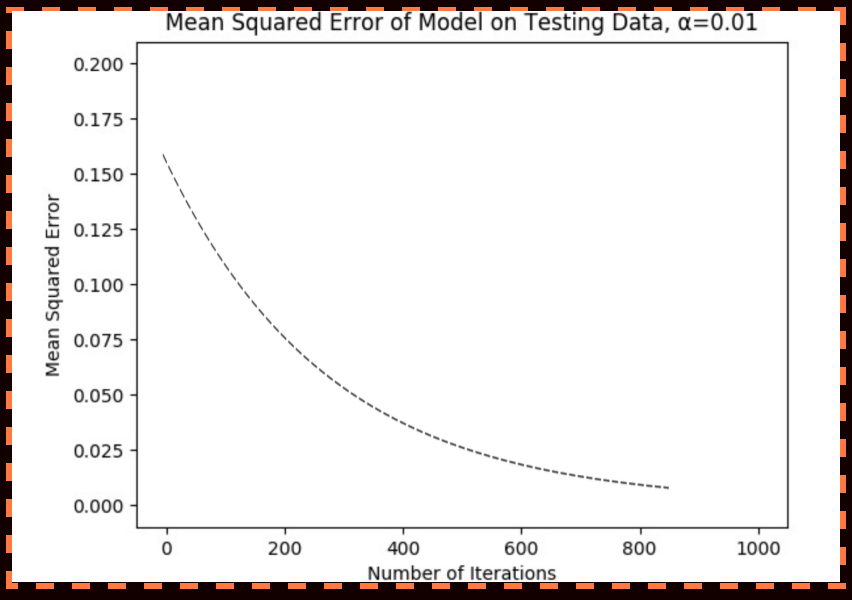



*Figure 12: A plot of model Mean Squared Error (MSE) as the number of iterations increases, α = 0.01, number of epochs = 1000, cutoff = 4 (optimal hyperparameter optimization)*

As can be seen in Figures 10-12, a learning rate ($\alpha$) of 0.01 and 1500 epochs are the hyperparameter values that I finalized for my model. Larger learning rates could have resulted in unstable training and tiny rates could have resulted in a failure to train. Figure 11 displays my finalized learning rate values: $\alpha$ = 0.01 and the number of epochs = 1500.

Overall, CountBOTula performs better on smaller maps than larger maps, especially after hyperparameter optimization. This could possibly be because the dataset my model was trained on does not include any examples involving larger than 13 x 13 boards. More specifically, the examples of my dataset include largely one map type: *empty_room.txt*, which could be another shortcoming of my implementation.

## SHORTCOMINGS + IMPROVEMENTS

There are shortcomings to CountBOTula along with possible improvements that I would attempt to implement if I had more time. One shortcoming is that CountBOTula does not perform as well on maps with central blocks, such as *center_block.txt* - it wins fewer times on these maps compared to *empty_room.txt*. This is because a wall-hugging strategy would be most effective in these types of maps, and CountBOTula does not incorporate a wall-hugging strategy. For example, CountBOTula performs slightly worse against WallBot on *center_block.txt* than on *empty_room.txt* which makes sense since WallBot employs a wall-hugging strategy so it performs relatively better on this type of map. If I had more

time, I would add a condition in my main *eval_func()* that would tell my bot to traverse the walls if it is playing a game on maps with center blocks. Currently as is, CountBOTula runs into barriers more often on *center_block.txt* than it does on *empty_room.txt* since there are more barriers present in *center_block.txt* than *empty_room.txt*.

In addition, CountBOTula could certainly improve the values of the weights produced by the supervised learning heuristic. If I had more time, I would ideally like to train my machine learning model on a larger dataset, perhaps 30,000 to 60,000 examples (as opposed to 5,750 examples). This is because a larger dataset would expose my model to more scenarios (giving it more “knowledge” while it trains). I would preserve the 80/20 train/test split I used for the data (in order to prevent overfitting to the training set), but more examples would be included in both the training and testing data. My weights might also improve if I multiply the regression value outputted by the *predict()* function in *tronmodel.py* by a calculated coefficient after training my model for several thousands of epochs. I made sure to normalize the preprocessed data to return numbers in the range 0 to 1. It is useful to scale the input attributes for a model that relies on the magnitude of values.