Hunter King B00528551
Alex Rudiuk B00672069

CSCI 6505 Assignment #7

Intro

This assignment explores and compares reinforcement learning through the use of a look-up table, and a function approximator as a means to learn to play tic-tac-toe and "four in a row". Initially, the system was trained against itself using the Q-learning algorithm as a means of training. Once the system was trained, a second system composed of a function approximator that used a sigmoidal multilayer perceptron was also trained. This second system used the same Q-learning algorithm to update the weights of the function approximator, but also used a back propagation method to update the weights once each game was completed.
It should be noted, that if an individual knows how to play tic-tac-toe well they should never lose due to a rather limited number of ways to play. This then extends such that if two players play, who both know how to play, the result will always be a tie. This also extends to "four-in-a-row"

Methods

Q-learning Tic-Tac-Toe

The first method used for tic-tac-toe was the Q-learning algorithm utilizing a lookup table. The lookup table stores the value of every potential board state and action. These values were stored in a 10 dimensional array (9 for board, 1 for action) such that the system can quickly point to an index and retrieve the corresponding value of the state instead of searching through a list for the state in question. While this method requires a significant amount of memory, it also allows for a significant increase in training (and testing) speed. The lookup table was initialized with random values ranging from $[-0.5 \rightarrow 0.5]$. Once the system was initialized, it was trained by playing against itself for 3 million games, updating the table through the Q-learning method (equation 1).

$$Q(s,a) \leftarrow Q(s,a) + \mu(r + \gamma max_{a'} Q(s',a') - Q(s,a)) \tag{1}$$

The results of the best train are shown in figure 1. The plot on the top shows an average of $\sim$32% wins. Meaning that 32/100 games end in not a tie. This imperfect result is due to our ϵ (random action selection) which we set to 0.1. The plot on the bottom shows the results of the same trained system, but with ϵ=0, playing against a version of itself with ϵ=0.1. The output is 2.0 whenever the player with ϵ=0 wins, and is 1.0 when the player with ϵ=0.1 wins. This proves that our system has learned to properly play the game as both players mostly tie, however due to the 10% chance of a random move by the opponent, the player sometimes wins (though never loses).

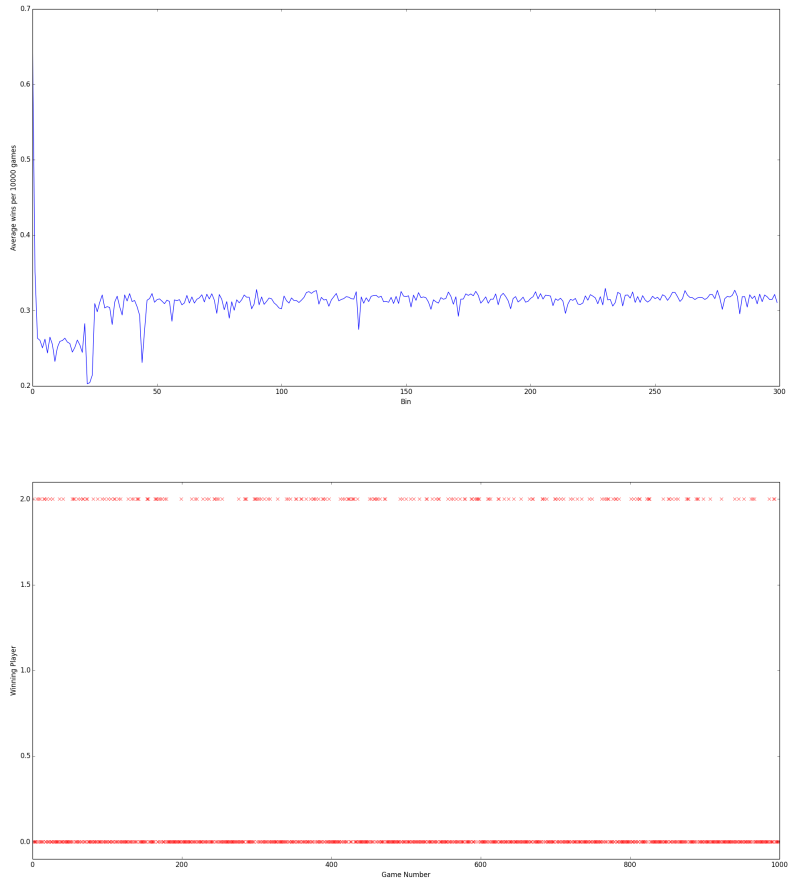

Figure 1: Top: Average results over training period, bins are 10000 games. Bottom: Trained system playing against a trained system with 10% chance to make random move.

Q-learning Four-in-a-row

Although Q-learning worked very well for tic-tac-toe, it unfortunately did not scale very well. Using Q-learning for tic-tac-toe with a lookup table required 3^9 states to be stored, and due to the nature of the method used, this was stored 3 times over. This equated to a requirement of 4MB of memory to store. When this is scaled up to the 3^{16} positions required for the four-in-a-row game it required 24GB of memory. We unfortunately did not have access to a computer with that amount of RAM so we could not play the game with our lookup table algorithm.

Function approximator

A solution to the problem of poor scaling (i.e. the "curse of dimensionality") is to use a function approximator to replace the lookup table. This function approximator uses a multilayer perceptron with a single hidden layer. The inputs correspond to the board state and the selected action (move). This requires only 28 inputs for tic-tac-toe and 49 inputs for four-in-a-row (see figure 2). The inputs are composed of 3 inputs for each potential state of a square on the board and 1 final input for the action (move) to be made. The three board location inputs correspond to either: player one in location (100), player 2 in location (010), or no player in location (001). Memory usage is significantly reduced

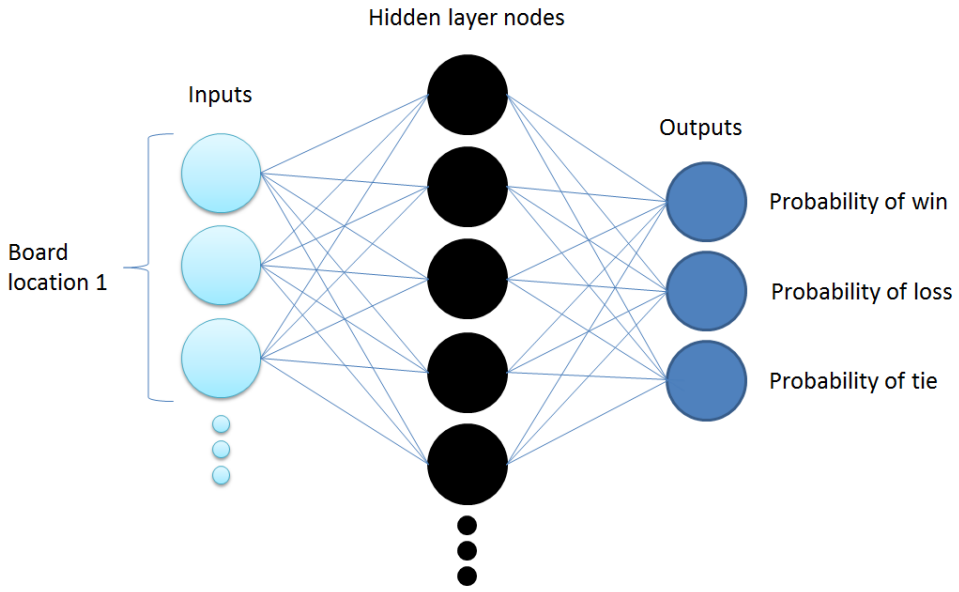

Figure 2: Multilayer perceptron used as function approximator

with a function approximator compared to a lookup table. A function approximator only needs to store the weights between nodes, inputs, and ouputs. Therefore, for tic-tac-toe with 40 hidden nodes used (40 was the optimal number of nodes selected by Gerald Tesauro for use in TD learning for backgammon), the system would only need to store 1240 weights. This is a significant reduction from the ∼19000 states stored by the lookup table. The function approximator scales $O(n)$, where the lookup table scales $O(3^n)$. However, the function approximator is not better in every way than the lookup table. Since the lookup table stores the specific value of a board state, and the function approximator works as it sounds (approximates this function), it is not fair to expect a perfectly trained system however the results for the best train are shown in figure 3. For the function approximator results shown in the graph we used td-λ where we remembered

all the past moves performed. Then when the game ends, we propagate the results back with a decay of γ applied every step back from the final reward. Otherwise, it was the same the Q-learning algorithm showed above. The algorithm does not perform as well as using a lookup table, but it was showing signs of learning (as shown by the decreasing curve). When we played against it, it was able to go for wins, but was less smart about stopping a loss. The current criteria we use for deciding a move is to pick the largest move which has a higher chance of winning vs losing, and then if none are available do the best drawing move. Unfortunatly, we were not able to try many variations on this formula due to time constraints, but we think that if we kept training the algorithm shown in figure 3 for many more million trials it would start to perform very close to a lookup table.

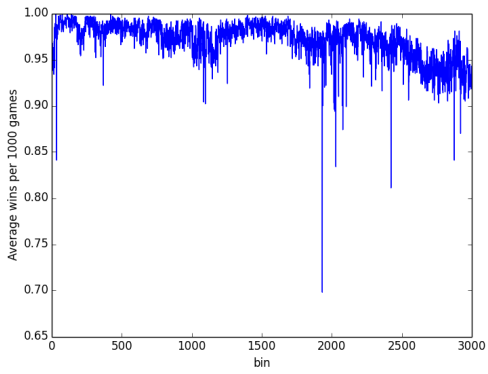

Figure 3: The average wins is decreasing over time, but it is decreasing much slower than using a lookup table.

Hunter King B00528551
Alex Rudiuk B00672069

Conclusions

The lookup table worked very well, and it probably achieved a perfect strategy before 3 million wins, we estimate around 500000 wins going by where the plot on the top of figure 1 stabilizes. We were not able to verify this empirically due to time constraints, but if the average amount of wins does not change in the graph it is likely not learning much anymore. It'd also likely work for a 4x4 board if we had enough ram to run the program. The function approximator saved a lot of memory because we do not need to store every state, instead we just store the weights for a perceptron. Unfortunatly, we were not able to achieve results as good as a lookup table using a function approximator. We did find that the algorithm was learning however, so if we trained longer it could possible be as good as the lookup table. One of the reasons it might be training so slow, is that if the program makes a bad move in the future, because of the epsilon term forcing it to randomly move, it punishes possibly good previous states. Future work would involve testing out a version of function approximation closer to Q-learning above where we estimate the optimal future reward instead of waiting till the end to update all the previous states. Or we come up with a more clever way of updating past states where we do not punish possible good states for a bad future state.