Overview

In this paper, I will outline the my implementation of a Tron bot that mixes two approaches, namely Adversarial Search and Machine Learning, describing the architecture and model sufficiently in detailed so that the reader could replicate my bot. Furthermore, I would chronicle my progress as well as explain the motivations behind my implementation. Finally, I would discuss further improvements that could be done.

The Back Story and Motivation

My first implementation consisted of envisioning the Tron problem as an Adversarial Search problem. The motivation behind it was that the Tron problem, as given to us, was two-player, sequential-move, constant-sum game modeled as an adversarial search problem. To solve it, I used an Alpha-Beta-cutoff algorithm with a Voronoi heuristic, which consists of the difference between the Voronoi regions between two players. having a cut-off depth of 5, this bot was able to beat Random and Wall bots at least 95% on any of the maps. However, while being efficient in the mid and end game stages, Voronoi heuristic fails to take into account the articulation points, which worsens its performance [1].
The problem then consists of finding a better value function for the beginning of the game. This can be done using supervised learning. Having found that function we can then produce a new heuristic, which will consist of a mix of Voronoi heuristic and a new learned value function

Implementation

To use supervised learning, I first had to generate some data. For each of the 100 iterations, I ran a game of Tron consisting of the TA bot 2 playing against my bot 10 times on an empty map. I used an empty map for training because the barriers produced by the player can mimic the walls of other maps. The decisions made by my bot are according to the previous model (initialized to the voronoi and 0 action value function) but also accounting for exploration with probability $\epsilon = 0.2$ During each game, for each decision that my bot took, I recorder the state, the action, the reward and the next state using a class called buffer. Note that in this case, the next state is not the state of the game after my bot made a move but rather the state that we find ourselves once the opponent made an action after our initial action. In my implementation, the state is characterized by the board and player locations.
Buffer: I implemented a separate class called Buffer for more consistent recording of data. It uses numpy arrays to store the board as 2D matrix where walls and barriers are represented by zero while everything else is represented by 0; locations of players for each state, the rewards and the undertaken action. It is also characterized by capacity (the number of records it must hold) as well as the batch size, the size of data on which we will learn.

Once the data has been generated, For the supervised learning part, I first wanted to use a Multi-Layered-Perceptron (MLP) using the python torch library. The goal of this MLP is to learn the action value of a state and an action. The way of feeding the data consisted of picking uniformly at random the batch size amount of samples from the data stored in the buffer. The first layer of the MLP was divided into three parts: first part took the board representation as input; second the location, third the action. The was an additional layer for the board representation, motivated by the belief that the board carries a lot of important information. The forwarding uses the Rectified Linear Unit function as the activation function, simply for the reason that it was simpler for me to comprehend. The output of those layers are the unified and fed to the final layer. To learn, we would compute the mean squared error and propogate it backwards in our MLP. We would then update the police with Adam optimization, which is described to be a more efficient approach than the stochastic gradient descent. After finishing the training, we save the model as trained.mlp.
With the function approximation done, my evaluation function that is called in Alpha-Beta-cutoff then uses the result from the voronoi heuristic and the function approximation by weighting their respective value

$$\mathsf{eval(state)} = (1-\alpha) \times \mathsf{Voronoi(state)} + \alpha \times \mathsf{max}\ Q(s,a)$$

.
Unfortunately, this approach threw me a number of errors which I couldn't debug after many hours of trying. Thus, I decided to use only a single layer, keeping everything the same but removing the second layer used for the board. This, for some reason fixed the bugs I was experiencing and was able to produce good results, ameliorating the simple Voronoi heuristic implementation from $5-10\%$ when playing against TA bot 2 on all maps. The optimal α I found was 0.1 and the cut off depth is 4.

Short Comings and Future improvements

Unfortunately, my algorithm only works for maps of size 13×13. Any other size of the board would cause my algorithm to fail. This is due to the fact that my learning does not generalize to other sizes. Thus, one improvement that could be made is the generalization to the dimension of any board. One approach could be to fix a certain size and transform board of other sizes into the fixed size in a way such that no information is lost.
Another possible improvement is to add additional layers as this might produce a more efficient model. If I had a bit more time, this would probably the next thing to do.