Behavioral Cloning vs Apprenticeship learning via IRL applied Tic-Tac-Toe

Cp468 project report

Adnan Badri: 200664980

Wilfrid Laurier University
Canada
07/19/2022

CP468 Project Report

Adnan Badri

July 2022

Contents

1 Introduction

Assume that you are able to observe two expert tic-tac-toe players playing a game and you wish to learn from their demonstrations. However, the catch is, you don't know the goal of either player (you just see them making moves and don't know why, ie you don't know the utility function they are trying to maximize). This project aims to discuss two algorithms that given such circumstances will output an optimal policy for the x player (it's the one that goes first).

2 Literature review

2.1 Comparison

In the real world, it is most of the time very difficult if not impossible to come up with utility functions that describe a task perfectly. Take for example the task of driving, it is very hard to come up with a utility function that describes "good/desirable" driving due to the fact that many many different factors are at play when driving. You have to balance between driving safely, comfort, speed, etc. in other words you have to weight all of those factors and balance between them. determining those weights (how more important speed is than comfort) is very difficult. Due to that, researchers have been very interested in developing algorithms that recover such complex utility functions when given expert demonstrations of the task. I will discuss two algorithms, one of which tries to recover the utility function and the other just mimics the expert demonstrations.

2.2 Max-margin

Inverse reinforcement learning is the problem of given observed behaviour of an agent inferring what it's reward/utility function must have been [2]. The first algorithm i will discuss is called **max-margin**. It is an inverse reinforcement learning algorithm (IRL), it doesn't have the features other more advanced IRL algorithms have but given that the following assumptions are satisfied, it recovers the utility/reward function and (because that is the best/most robust descriptor of a task) also outputs a policy π that is at least as good as the policy the expert demonstrations came from.

1. you are Given the MDP $M = \{A, S, \gamma\}$.
2. you can run your agent in the environment.
3. You are also given the expert trajectories.
4. the utility/reward function can be represented as a linear combinations of the features ie $R(s) = W^T * (\phi(s)$ where $R(s)$ is the reward at state s and $\phi(s)$ is a feature extractor, that given a state returns a vector denoting its features.

The max-margin algorithm does this by finding a reward function parameterized by W that makes the expert policy (the one the demonstrations came from) look better than any other policy by the highest margin [1].

2.3 Behavioral cloning

The second algorithm i will discus is behavioral cloning. It too relies on expert demonstrations, but instead of trying to recover the utility function it instead uses the expert demonstrations and runs them through a classifier. That is, it takes the states as the features, and the actions the demonstrator took in those states as the label. It then runs a normal classification algorithm on that. This too outputs a policy that is good if the number of demonstrations is large enough. the following table shows a comparison of the two algorithms discussed above.

Table 1: Inverse reinforcement learning vs behavioral cloning

Algorithm	Pros	Cons
IRL	- Has the potential to surpass demonstrator - Transferable to other envs - needs less data data than Behavioral cloning - Can even learn from bad demonstrator - Can be used when coming up with a utility function is very hard	- There are MANY MANY different reward funcs, how do you find the right one???? -VERY computationally costly
Behavioral cloning	- Super easy (just a normal classifier) - Relatively efficient/fast	- Can never get better than demonstrator - Needs huge amounts of data - Gets in trouble if input is not i.i.d - Can't be used in another env

3 Project Description

Now, that we have discussed the literature the point of this project becomes applying those 2 different algorithms in an environment that is adversarial in nature. That is because all of the research papers/tutorials only apply those algorithms to simple environments like Gridworld. So, the rest of the report will talk about applying the algorithms to the game of tic-tac-toe. So, the goal becomes checking if those algorithms perform well in the tic-tac-toe game.

3.1 Data Collection

like any other machine learning algorithm, the algorithms presented here require Data. However, there is no public data that we can use for training(no tic-tac-toe games online) so we will have to **generate** our own data. We can do this by using the **MiniMax** algorithm for both the x and o players and recording the different states the x player sees and the actions it takes in those states. So, as a result we will get expert trajectories $D = [[(s_0, a_0), (s_1, a_1), ...(s_i, a_i)], [(s_0, a_0), (s_1, a_1), ...(s_i, a_i)], ...]$
where each sublist of D is an episode of the game and a_i is the action player x took in state s_i. By starting x at each of the 9 spots and following **MiniMax** afterwards we can get 9 different episodes of the game, thus making the length of D 9. So, an example of what an episode in D would look like is as follows. notice how when "x" moves, "o" also moves and the next time "x" sees the board, there are two new spots that are drawn on. $D[0] = \begin{array}{c|c|c} & & \\ \hline & & \\ \hline & & \end{array}$,(0,0) , $\begin{array}{c|c|c} \text{x} & & \\ \hline & \text{o} & \\ \hline & & \end{array}$,(1,0),......., $\begin{array}{c|c|c} \text{x} & \text{o} & \text{x} \\ \hline \text{x} & \text{o} & \text{o} \\ \hline \text{o} & \text{x} & \end{array}$,(2,2)
the tuple next to the board represents the location "x" placed its mark on so (0,0) means top left.

4 State representation/Feature selection

assume we have the following board $\begin{array}{c|c|c} \text{x} & \text{o} & \text{x} \\ \hline \text{x} & \text{o} & \text{o} \\ \hline \text{o} & \text{x} & \end{array}$ if we replace the "x" with 1 and the "o" with 2 and the empty spaces with 0 we get the new matrix

$$\begin{pmatrix} 1 & 2 & 1 \\ 1 & 2 & 2 \\ 2 & 1 & 0 \end{pmatrix}$$

if we flatten that matrix we get the following vector $[1, 2, 1, 1, 2, 2, 2, 1, 0]$, that will be our state representation method. As for representing actions, just use a tuple where the first element is the row of the square and the second element is the col of the square so $action = (row, col)$ where both row and col start at 0. in other words $\phi(s)$ would be the vector defined above, where s is a state.

5 Methodology

5.1 Introduction to languages

The language *python* was used to develop all of the algorithms discussed above.

5.2 Supporting packages

the numpy package was extensively used throughout this project to do the matrix math required for the max-margin algorithm. It was also used to do certain actions with a certain probability.

5.3 Constraints

Due to the fact that the max-margin algorithm is a very computationally expensive one (as shown in the Implementation section), the algorithm was only run for 100 iterations. Also, the RL solver algorithm (returns a policy given a reward function) only took 30,000 steps to find the optimal policy. this is also due to the computational cost of solving the forward reinforcement learning. Due to all this, the results will probably be affected.

5.4 Assumptions

For the Max-margin algorithm to work effectively, the list given in section 2.2 is assumed to be true. For the behavioral cloning algorithm to work as excepted, the demonstrations should be coming from an optimal demonstrator (true since Minimax is used).

5.5 Ml algorithm discussion

5.5.1 Maxmargin

Since we assumed the reward was a linear combination of the features $\phi(s)$, the Maxmargin algorithm would pick W such that the policy under it visits the same states as the optimal policy (same feature expectation to insure that it does at least as good as the optimal policy) and it makes the optimal policy look better than any other policy by the highest margin (partially eliminates the ambiguity of W). In other words, Maxmargin finds a W that the policy under it performs comparable to the expert policy and it explains the actions of the demonstrator (optimal policy) the best. To solve the forward reinforcement learning problem (given reward function find optimal policy), tabular Q-Learning was used with epsilon greed where ϵ is 0.2 (agent picks random action 20 percent of the time to explore environment).

5.5.2 Behavioral cloning

Since this is just a normal supervised classification problem the fact that the expert trajectories D are grouped into "episodes" can be ignored and only the state actions pairs just be looked at instead. for each state action pair, the state is the feature (ie get $\phi(state)$) and the action is the label. this defines an easy classification problem. Something like a neural network can be used for this. **However this algorithm will not be implemented since $|D|$ is only 9, and classification algorithms require a lot more data than that. so, no point in wasting time developing it (its actually implemented in the code files but not analyzed)**.

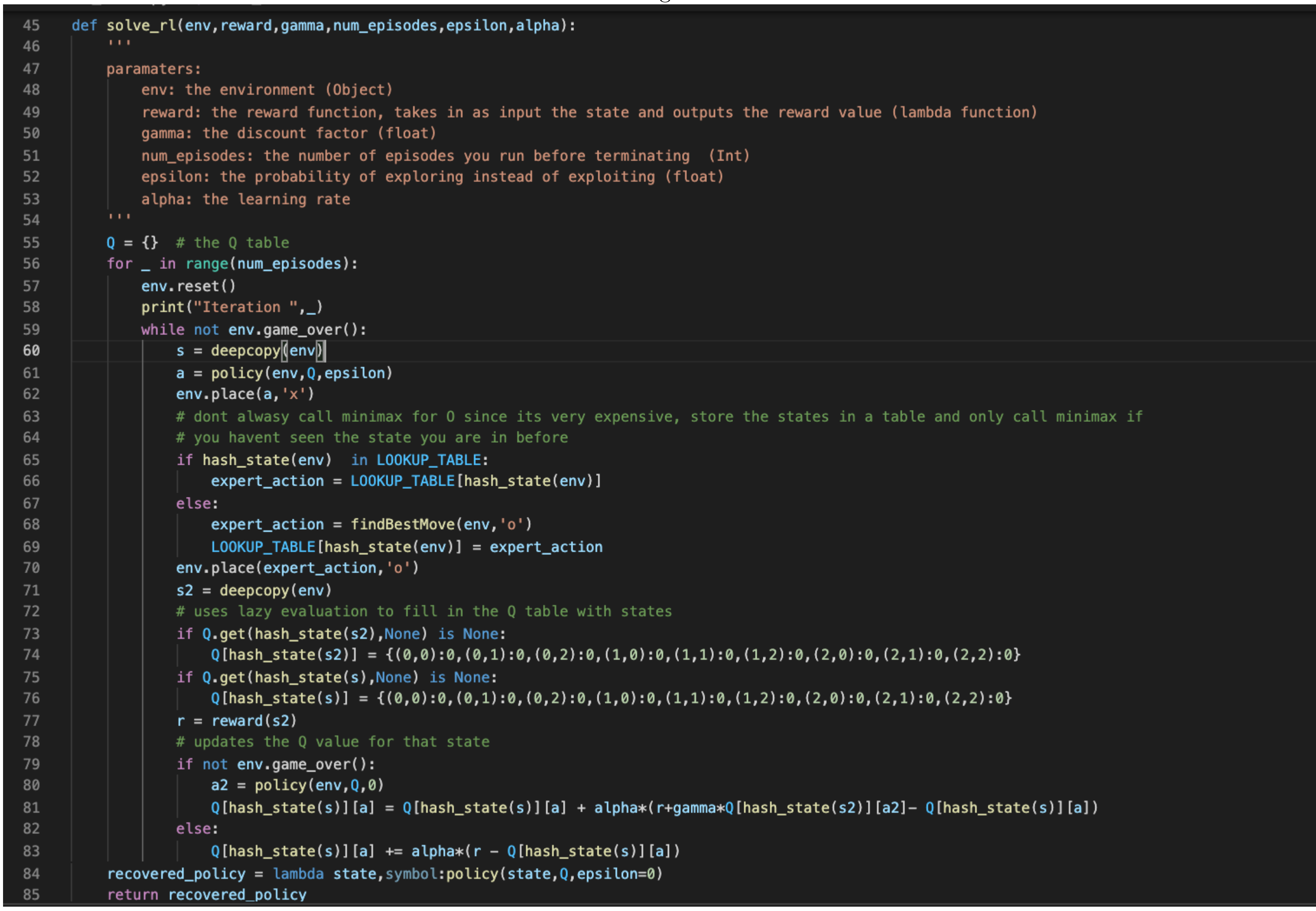
5.6 Algorithm implementation

The Maxmargin algorithm uses 2 other auxiliary algorithms to accomplish its task, mainly the Minimax algorithm (to generate D) and the forward reinforcement learning algorithm (to find policy given reward function). So, the forward RL algorithm will be discussed first before discussing the implementation of the Maxmargin algorithm (Minimax is too simple so it won't be discussed).

5.6.1 Forward RL

This algorithm uses epsilon-greedy to explore the environment/game in conjunction with Tabular-Qlearning. it's implementation is given bellow

Code for the forward RL algorithm



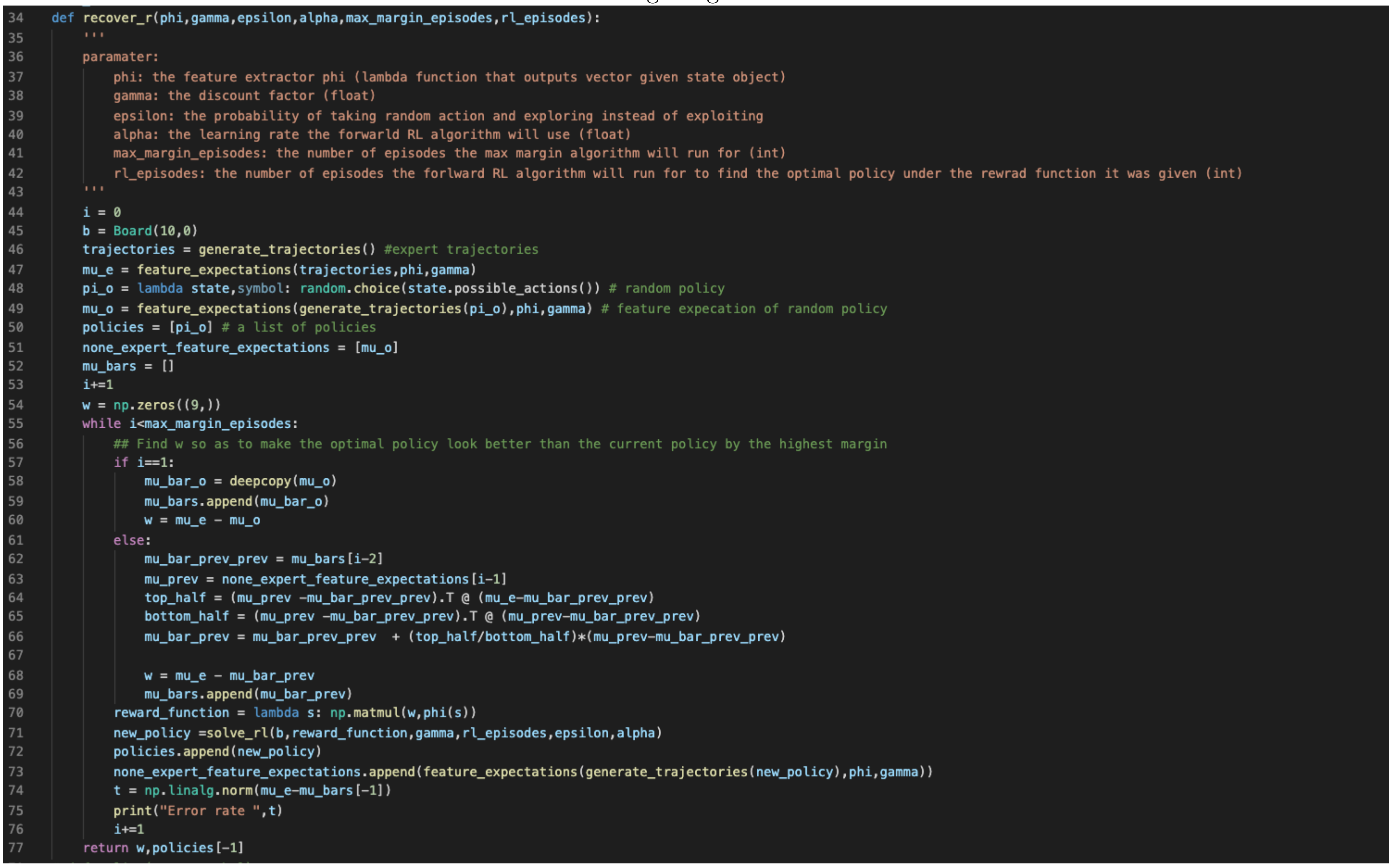
```
45  def solve_rl(env,reward,gamma,num_episodes,epsilon,alpha):
46      '''
47      paramaters:
48          env: the environment (Object)
49          reward: the reward function, takes in as input the state and outputs the reward value (lambda function)
50          gamma: the discount factor (float)
51          num_episodes: the number of episodes you run before terminating  (Int)
52          epsilon: the probability of exploring instead of exploiting (float)
53          alpha: the learning rate
54      '''
55      Q = {}  # the Q table
56      for _ in range(num_episodes):
57          env.reset()
58          print("Iteration ",_)
59          while not env.game_over():
60              s = deepcopy(env)
61              a = policy(env,Q,epsilon)
62              env.place(a,'x')
63              # dont alwasy call minimax for O since its very expensive, store the states in a table and only call minimax if
64              # you havent seen the state you are in before
65              if hash_state(env)  in LOOKUP_TABLE:
66                  expert_action = LOOKUP_TABLE[hash_state(env)]
67              else:
68                  expert_action = findBestMove(env,'o')
69                  LOOKUP_TABLE[hash_state(env)] = expert_action
70              env.place(expert_action,'o')
71              s2 = deepcopy(env)
72              # uses lazy evaluation to fill in the Q table with states
73              if Q.get(hash_state(s2),None) is None:
74                  Q[hash_state(s2)] = {(0,0):0,(0,1):0,(0,2):0,(1,0):0,(1,1):0,(1,2):0,(2,0):0,(2,1):0,(2,2):0}
75              if Q.get(hash_state(s),None) is None:
76                  Q[hash_state(s)] = {(0,0):0,(0,1):0,(0,2):0,(1,0):0,(1,1):0,(1,2):0,(2,0):0,(2,1):0,(2,2):0}
77              r = reward(s2)
78              # updates the Q value for that state
79              if not env.game_over():
80                  a2 = policy(env,Q,0)
81                  Q[hash_state(s)][a] = Q[hash_state(s)][a] + alpha*(r+gamma*Q[hash_state(s2)][a2]- Q[hash_state(s)][a])
82              else:
83                  Q[hash_state(s)][a] += alpha*(r - Q[hash_state(s)][a])
84      recovered_policy = lambda state,symbol:policy(state,Q,epsilon=0)
85      return recovered_policy
```

lines 60-61 record the state and action 'X' took and lines 65-71 record the action 'O' took and the new state that lead to. 73-83 update the Q-value based on s(state 'X' was in),a(action it took in that state), s2 (the new state it saw when it was it's turn again) and a2(the optimal action it took in that state). "policy" is just a function that given an epsilon, a state, and a Q table picks the action with the highest Q-value (1-ϵ) of the time and a random action ϵ of the time.

5.7 Max-margin

The block of code bellow implements the algorithm from this paper [1].

Code for the Max-margin algorithm



```
34  def recover_r(phi,gamma,epsilon,alpha,max_margin_episodes,rl_episodes):
35      '''
36      paramater:
37          phi: the feature extractor phi (lambda function that outputs vector given state object)
38          gamma: the discount factor (float)
39          epsilon: the probability of taking random action and exploring instead of exploiting
40          alpha: the learning rate the forwarld RL algorithm will use (float)
41          max_margin_episodes: the number of episodes the max margin algorithm will run for (int)
42          rl_episodes: the number of episodes the forlward RL algorithm will run for to find the optimal policy under the rewrad function it was given (int)
43      '''
44      i = 0
45      b = Board(10,0)
46      trajectories = generate_trajectories() #expert trajectories
47      mu_e = feature_expectations(trajectories,phi,gamma)
48      pi_o = lambda state,symbol: random.choice(state.possible_actions()) # random policy
49      mu_o = feature_expectations(generate_trajectories(pi_o),phi,gamma) # feature expecation of random policy
50      policies = [pi_o] # a list of policies
51      none_expert_feature_expectations = [mu_o]
52      mu_bars = []
53      i+=1
54      w = np.zeros((9,))
55      while i<max_margin_episodes:
56          ## Find w so as to make the optimal policy look better than the current policy by the highest margin
57          if i==1:
58              mu_bar_o = deepcopy(mu_o)
59              mu_bars.append(mu_bar_o)
60              w = mu_e - mu_o
61          else:
62              mu_bar_prev_prev = mu_bars[i-2]
63              mu_prev = none_expert_feature_expectations[i-1]
64              top_half = (mu_prev -mu_bar_prev_prev).T @ (mu_e-mu_bar_prev_prev)
65              bottom_half = (mu_prev -mu_bar_prev_prev).T @ (mu_prev-mu_bar_prev_prev)
66              mu_bar_prev = mu_bar_prev_prev  + (top_half/bottom_half)*(mu_prev-mu_bar_prev_prev)
67
68              w = mu_e - mu_bar_prev
69              mu_bars.append(mu_bar_prev)
70          reward_function = lambda s: np.matmul(w,phi(s))
71          new_policy =solve_rl(b,reward_function,gamma,rl_episodes,epsilon,alpha)
72          policies.append(new_policy)
73          none_expert_feature_expectations.append(feature_expectations(generate_trajectories(new_policy),phi,gamma))
74          t = np.linalg.norm(mu_e-mu_bars[-1])
75          print("Error rate ",t)
76          i+=1
77      return w,policies[-1]
```

the **generateTrajectories()** function generates trajectories for 'X' when it's following a certain policy (the policy is minimax by default), in other words it creates the trajectories D. lines 46-54 do the initialization. lines 55-69 pick W so as to make the expert look better than the current policy by the highest margin. this is done by comparing their expected rewards. line 73 appends the feature expectations of the current policy to a list so it can be compared to the feature expectations of the optimal policy later on.

6 Experimental analysis

The max-margin algorithm was run on an environment where the Demonstrations D and the opponent 'X' where both using the minimax algorithm and picked the first action if multiple actions had the same minimax value. the following list shows what the different hyper parameters where set to.

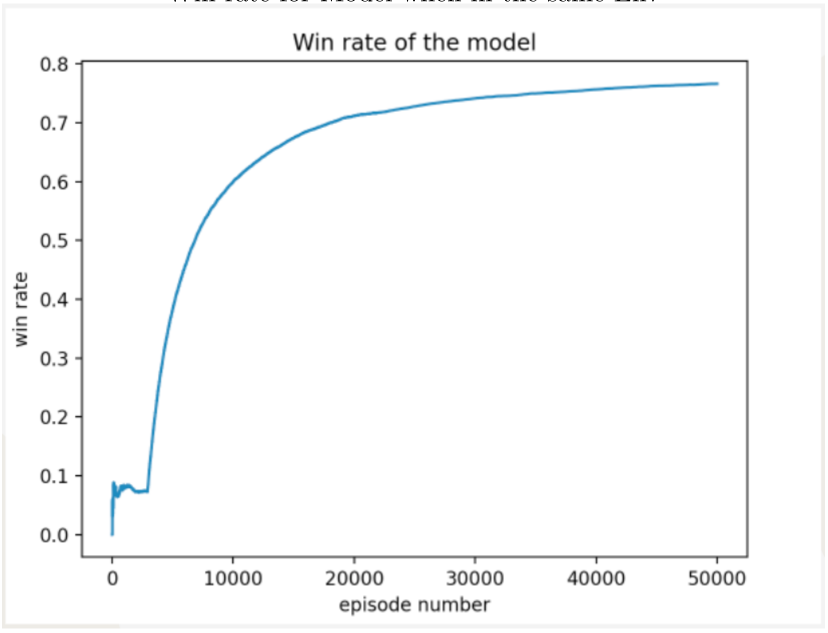
1. the discount factor γ was set to 0.9
2. the learning rate α was set to 0.1
3. the exploration probability ϵ was set to 0.2
4. numEpisodesMaxMargin was set to 100 because maxmargin is a very inefficient algorithm
5. numRLEpisodes was set to 30,000, that is the optimial policy for a reward function was estimated in 30,000 episodes (again because of computational cost)

Since one of the most attractive things about IRL algorithms is their ability to do transfer learning, this section will investigate how well the agent is able to adapt when put in an environment that has different dynamics than the one it was trained in

6.1 Optimal/same environment

in the following section we will discuss how the algorithm performs in the same environment the reward function was recovered in. The following plot shows the win-rate (where Win is defined to be a draw game)

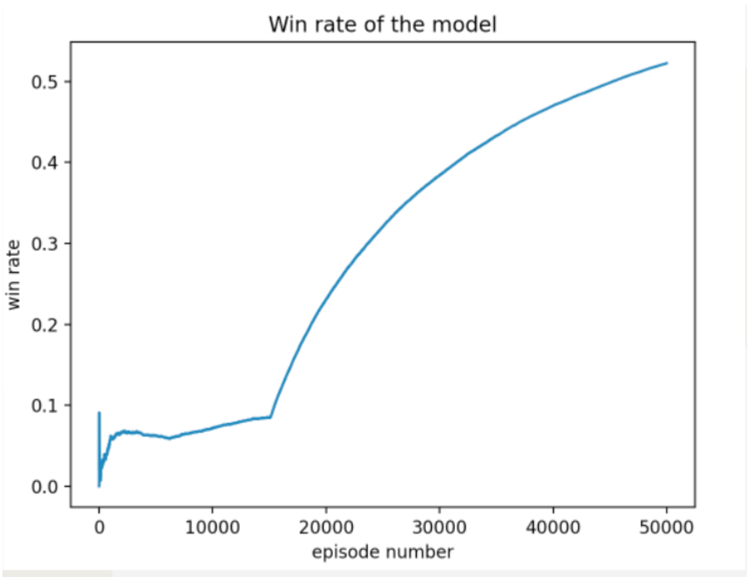
Win rate for Model when in the same Env



As we would expect, the model performed very well in the same environment the reward function was recovered in. Reaching as high as 78 percent

6.2 Environment that is a little different

in the following section we will discuss how the algorithm performs in an environment where the 'O' player still uses minimax but now picks the last action if it sees multiple actions with the same minimax value. a plot of the win rate of the model is shown bellow

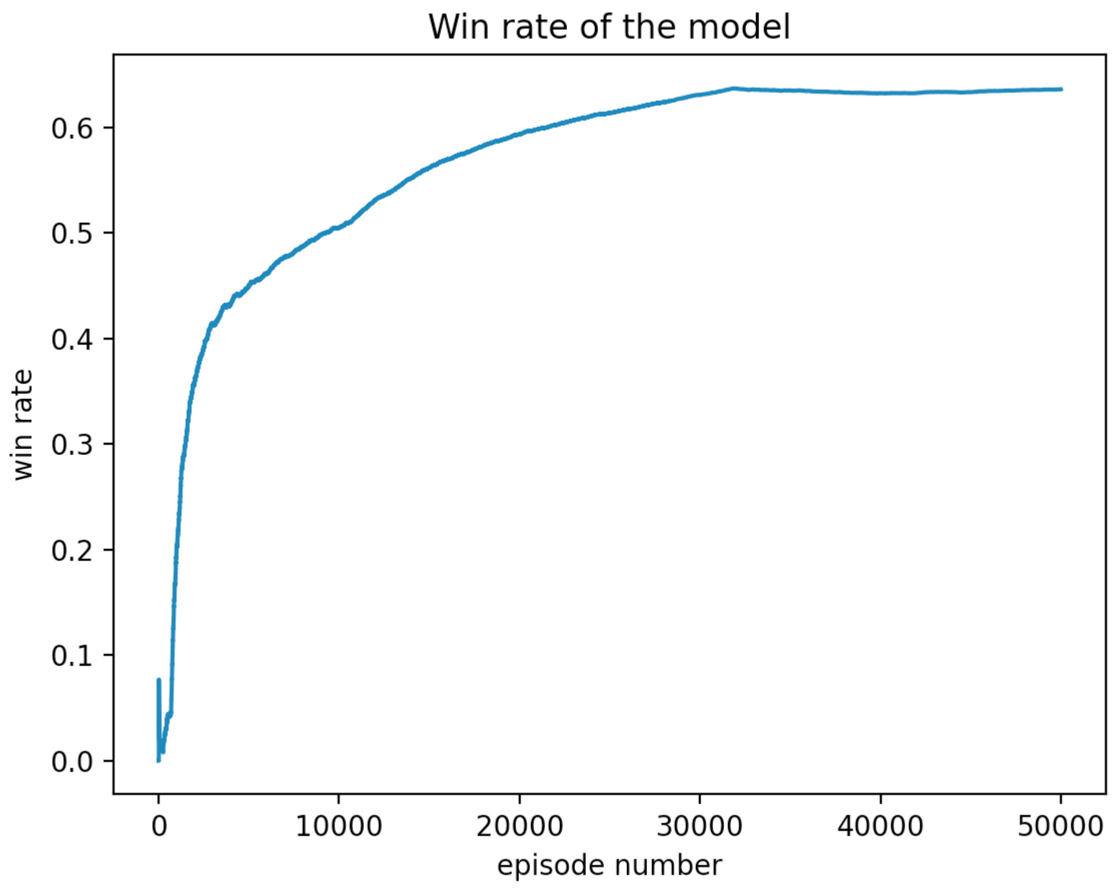


Win rate in new env

from the plot above we can deduce that the model is having difficulty adjusting to the new environment. It's win-rate dropped from 78 to just above 50. This can be took in 2 ways: first way is that max-margin can't really do transfer learning and the second way is that since the environment is different we should allow it to train for longer (maybe 100,000 steps instead of just 50,000).

6.3 Environment that is very different

Now, the environment in the above section was only a little bit different from the environment the agent was trained on. What if it was more different? what if the 'O' player still played using minimax but when encountered with states with the same minimax value it picked **randomly** between them? This section will investigate the performance of the Agent when put in such an environment.



Win rate in new env

As the plot shows, the win rate actually got higher even though this environment is a lot more complex than the one above (has stochastic nature) the win rate actually got higher. even going as high as 64 percent. From this we can conclude that the bot trained in this environment can play against any optimal tic-tac-toe player and get to a draw about 64 percent of the time. Which is very impressive.

7 Feature improvements

It would be nice if the performance of the agent could be improved, and in fact that is the case. The bellow list lists a couple of things that might have been affecting the performance of the agent and when changed will lead to a better agent

1. the reward function was assumed to be linear in $\phi(s)$, what if it wasn't or at least not perfectly? there are other IRL algorithms that can be applied in that case and they might increase the performnance of the agent.
2. what if we trained for more? the training time was very little, so training it and leaving it for a day might yield very good results
3. what if we used a more advanced IRL algorithm? max-margin is to IRL algorithms what linear regression is to regression algorithms. i.e it's as basic as you can get, so perhaps applying a more advanced algorithm would yield better results.

and so on and so forth.

8 Feature analysis

we only tested the agent in 3 environments, the following list gives more tests that can be ran that might yield interesting results

1. what if in the demonstrations D the 'X' player acted optimally but the 'O' player didn't? that is, will the agent still be able to learn if it played against a dumb opponent? this is an interesting philosophical question: if you were never challenged, will you ever reach your true potential?

2. what if in the demonstrations the 'X' player didn't act optimally 100 percent of the time? will the agent still be able to learn (there are IRL algorithms that can handle sub-optimal 'experts')?

9 Conclusions

In conclusion, in a sense we accomplished our goal of making a tic-tac-toe agent that can learn from observing only 9 games. not only learn but have a win rate of 64 percent against **any** optimal player.

10 References

References

[1] Abbeel, P., and Ng, A. Y. (2004, July). Apprenticeship learning via inverse reinforcement learning. In Proceedings of the twenty-first international conference on Machine learning (p. 1).

[2] Arora, S., Doshi, P. (2021). A survey of inverse reinforcement learning: Challenges, methods and progress. Artificial Intelligence, 297, 103500.