By
Pushpa Latha Vudatha
Rohit Donepudi
Dec 10th 2021

Table of Contents

Introduction

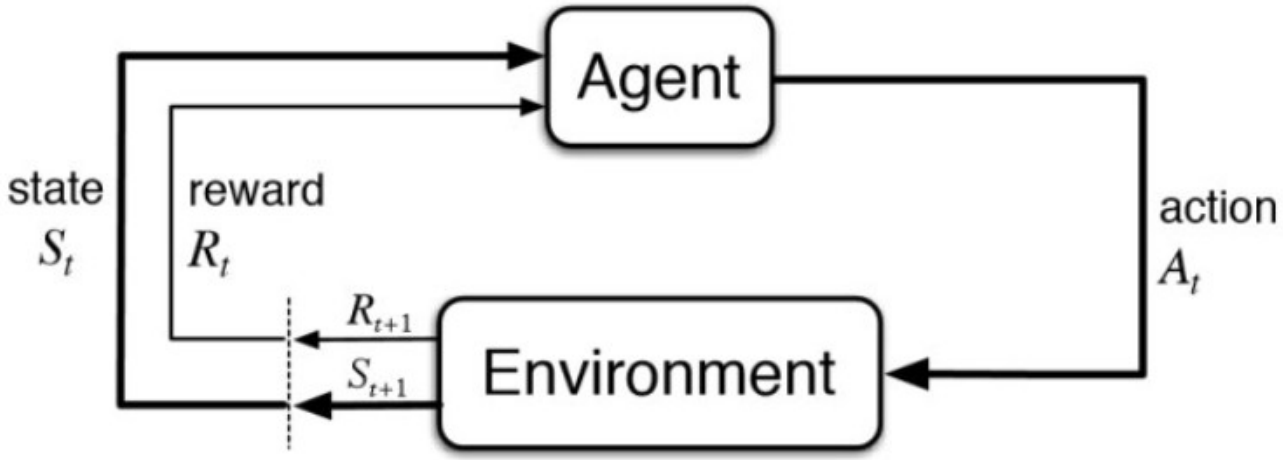
Tic-tac-toe is traditionally a popular board game among kids: in its 3 by 3 board two persons alternately place one piece at a time; one wins when he or she has three pieces of his or her own in a row, whether horizontally, vertically, or diagonally. This work will employ reinforcement learning methods in its version of “afterstate” evaluation to implement simple board games such as Tic-Tac-Toe.

In this project work we have implemented grid world game by iteratively updating Q value function, which is the estimating value of (state, action) pair. This time let’s look into how to leverage reinforcement learning in adversarial game — tic-tac-toe, where there are more states and actions and most importantly, there is an opponent playing against our agent.

Reinforcement Learning

Say, we have an agent in an unknown environment and this agent can obtain some rewards by interacting with the environment. The agent ought to take actions so as to maximize cumulative rewards. In reality, the scenario could be a bot playing a game to achieve high scores, or a robot trying to complete physical tasks with physical items; and not just limited to these.

The goal of Reinforcement Learning (RL) is to learn a good strategy for the agent from experimental trials and relative simple feedback received. With the optimal strategy, the agent is capable to actively adapt to the environment to maximize future rewards.



As shown in the figure above, the reinforcement learning framework comprises the following elements

- Agent - Entity learning about the environment and making decisions. We need to specify a learning algorithm for the agent that allows it to learn a policy.
- Environment - Everything outside the agent, including other agents
- Rewards - Numerical quantities that represent feedback from the environment that an agent tries to maximize

    - Goal reward representation: 1 for goal, 0 otherwise
    - Action penalty representation: -1 for not goal, 0 once goal is reached
- State: A representation of the environment. At time step ,the agent is in state where is the set of all possible states
- Action: At time step , an agent takes an action where is the set of actions available in state
- Policy: A policy tells the agent what action to take in a given state.

By considering the opponent as part of the environment which the agent can interact with, after certain amount iterations, the agent is able to planning ahead without any model of the agent or environment, or conducting any search of possible future actions or states. The advantage is obvious in that the method saves the effort of complex mathematical deduction or exploration of large number of search space, but it is able to achieve state-of-art skill by simply trial and learn.

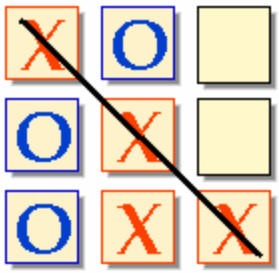
In the following sessions, we will:

1. Firstly, train 2 agents to play against each other and save their policy
2. Secondly, load the policy and make the agent to play against human

State Setting

Firstly, we need a State class to act as both board and judge. It has functions recording board state of both players and update state when either player takes an action. Meanwhile, it is able to judge the end of game and give reward to players accordingly.

To formulate this reinforcement learning problem, the most important thing is to be clear about the 3 major components — state, action, and reward. The state of this game is the board state of both the agent and its opponent, so we will initialize a 3x3 board with zeros indicating available positions and update positions with 1 if player 1 takes a move and -1 if player 2 takes a move. The action is what positions a player can choose based on the current board state. Reward is between 0 and 1 and is only given at the end of the game.



Multi-dimensional vectors are used to describe the state space of each situation. For example, as shown in Figure 1, we give it the following representing vector

s = [1, 2, 0, 2, 1, 0, 2, 1, 0] T

where 1 indicates player "X" places a piece in this location, 2 indicates player "O" places a piece in this location, and 0 indicates this is an empty location.

(2) Reinforcement learning for evaluation of afterstates

Reinforcement learning (RL) is an unsupervised machine learning technique, which "learns" from the interactive environment's rewards to approximate values of state-action pairs and maximize the long-term sum of rewards. It has four essential components: state set S, action set A, rewards from the environment R, and values for state-action pairs V.

A little different from the standard RL and specifically for board game applications, we combine state set S and action set A into a new "afterstate" set S. The reasons are: (1) in board games, a state after a move is deterministic; (2) Different "prestate" and action may come to the same "afterstate", thus possibly holding redundancy in many state-action pairs.
We will use temporal-difference method, one of reinforcement learning techniques, to approximate state values by updating values of visited states after each training game.

$$V(s) \leftarrow V(s) + \alpha [V(s') - V(s)]$$

where s is the current state, s' is the next state, V(s) is a state value for state s, and α is the learning rate, varying within (0, 1]

Training

For simple games such as Tic-tac-toe, two compute agents will play against each other and learn game strategies from simulated games. This training method is called self-play, which has several advantages such as that an agent has general strategies rather than those associated with a fixed opponent. Most of this project would stick to the generation of self-play training and the result shows its good performance though it might have a slow convergence problem in the early training stage.

Two phases in the game framework

The algorithm's framework consists of learning phase and game-play phase. Below is the brief description of the algorithm's structure:

a. In each episode of the learning phase

1. Observe a current board state s;
2. Make a next move based on the distribution of all available V(s') of next moves;
3. Record s' in a sequence;
4. If the game finishes, it updates the values of the visited states in the sequence and starts over again; otherwise, go to 1).

b. The game-play phase makes a "greedy" decision based on the learned state values. Every time

the computer is making next move,

1. Observe a current board state s;
2. Make a next move based on the distribution of all available V(s') of next moves;
3. Until the game is over and it starts over again; otherwise, go to 1).

It should be noted that these online afterward learning may not be included in Step 3) of the game-play phase as long as the game strategies are considered to be solid.

Algorithm

To solve this problem I will create a State class that represents the board as a player sees it. Each of the two players retain their own copy of the board. This is possibly an inefficient design, but this is what I will run with.

Given tic-tac-toe is a 2 player game, I essentially simulate two different environments. In the first one, the agent is player X and in the second one the agent is player Y.

After the agent plays, the move by the opposing player is considered a change in the environment resulting from the agent’s actions. The new state the agent lands in is a board where the opposing player has already made his/her move.

State Functions

Refer state.py

State - Class represnting the borad state. The class has methods to update the state vector, applying rewards, actions that are performed during the play.

getHash() - get unique hash of current grid state. The getHash function hashes the current board state so that it can be stored in the state-value dictionary.

When a player takes an action, its corresponding symbol will be filled in the board. And after the state being updated, the board will also update the current vacant positions on the board and feed it back to the next player in turn.

winner() - Calculate the rewards in each grid of the board at current state position and figures out the win player. After each action being taken by the player, we need a function to continuously check if the game has ended and if end, to judge the winner of the game and give reward to both players. The

winner function checks sum of rows, columns and diagonals, and return 1 if p1 wins, -1 if p2 wins, 0 if draw and None if the game is not yet ended. At the end of game, 1 is rewarded to winner and 0 to loser. One thing to notice is that we consider draw is also a bad end, so we give our agent p1 0.1 reward even the game is tie(one can try out different reward to see how the agents act).

availablePositions() - provides available postions for to make the next move

updateState () - Update the state value of the grid based on player symbol,

value = 1 for player 1

Value = -1 for player 2

giveReward () - Rewards are updated at the end of each game. A reward works like a feedback and lets the player to correct the game.

0 <= reward <= 1

Usually for the computer to play better we give reward 1 upon a win and 0 upon lose and vice versa for player 2 (ie that is be simulated like a human)

reset () - restarts the game

play() - this is the place where training happens. p1 and p2 playes the game iteratively and update their states with the rewards based on the win state. Now that our agent is able to learn by updating value estimation and our board is all set up, it is time to let two players play against each other.

During training, the process for each player is:

- Look for available positions
- Choose action
- Update board state and add the action to player’s states
- Judge if reach the end of the game and give reward accordingly

showGrid () - displays the board with player positions on the console.

Player Class

Refer player.py

We need a player class which represents our agent, and the player is able to:

- Choose actions based on current estimation of the states
- Record all the states of the game
- Update states-value estimation after each game

- Save and load the policy

chooseAction() - We store the hash of board state into state-value dict, and while exploitation, we hash the next board state and choose the action that returns the maximum value of next state.

feedbackReward() - To update value estimation of states, we will apply value iteration which is updated based on the formula below

$$V(S_t) \leftarrow V(S_t) + \alpha \left[V(S_{t+1}) - V(S_t) \right]$$

The formula simply tells us that the updated value of state t equals the current value of state t adding the difference between the value of next state and the value of current state, which is multiplied by a learning rate α (Given the reward of intermediate state is 0).

savePolicy & loadPolicy - At the end of the training(playing after certain amount of rounds), our agent is able to learn its policy which is stored in the state-value dict. We need to save this policy to play against a human player.

HumanPlayer - Now our agent is all set up, in the last step we need a human class to manage to play against the agent. This class includes only 1 usable function chooseAction which requires us to input the board position we hope to take.

Mostly the same, we let player 1(which is our agent) to play first, and at each step, the board is printed.

Stats Function

Refer ***ticTacToe_stats_generation.ipynb***

To verify how well the algorithm learn, we have designed a code that verifies the game between two players (ie) computers using the policies that are learn during training.

```
--------------------------------------------------------------------------------
                                      STATS
--------------------------------------------------------------------------------


               Player 1 started the match      : 515/1000
               Player 2 started the match      : 485/1000


               Player 1 won                    : 515/1000
               Player 2 won                    : 485/1000
               Tie                             : 0/1000

--------------------------------------------------------------------------------
```

Results

- Human can not beat the computer if the computer goes first.
- Player X represents the computer and Player O represents human.
- The more it training, the more sophisticated its game policies are.
- It seems like X typically wins around 60% of games.
- Computer wins more games than Human as it gets to start first and make more moves than Human.
- For both players, occupying the central square in the first move maximizes the chances of winning.
- The winning sequence is most likely to be along the diagonal, for both players

Conclusion

This project was essential to establish basics of Artificial Intelligence techniques, to be precise the reinforcement learning, Q-learning techniques. Overall, this may sound like a simple exercise, but we actually found it very challenging and fun. We learned a lot more about reinforcement learning and about writing a full game with multi-classes using python has been challenging too. We had to think about what rules we want, which chunk of code can become a function and find the bugs. Lastly, it is clear that reinforcement learning is incredibly powerful and full of endless possibilities and therefore it is key to continue having as much hands on experience as possible to get a better handle of it.

References

1. https://github.com/JaeDukSeo/reinforcement-learning-an-introduction
2. http://incompleteideas.net/book/the-book-2nd.html
3. https://www.govindgnair.com/post/solving-tic-tac-toe-with-reinforcement-learning/