Game AI Documentation

Zachary Marion

Dec 30, 2018

Getting Started:

GameAI contains a series of well-defined abstractions that are common in AI, such as a games, agents, and trainers which optimize the behavior of agents. As long as a class inherits from the base implemetation of a primitive (e.g. `Agent`) and implements the required methods, it can be used in place of the standard implementations given.

CHAPTER 1

Getting Started

1.1 Installation

Because this is still in alpha and under active development, it has not been released to PyPi. You can install via TestPyPi using the following command:

```
pip install --index-url https://test.pypi.org/simple/ gameai
```

1.2 Basic Example

```
from gameai.games import TicTacToe
from gameai.agents import RandomAgent, MCTSAgent
from gameai.core import Arena, Player

# Create our game
game = TicTacToe()

# Inititalize our agents
mcts_agent = MCTSAgent()
random_agent = RandomAgent()

# We train the mcts agent
mcts_agent.train(game, verbose=True, num_iters=10000, num_episodes=100)

player0 = Player(0, mcts_agent)
player1 = Player(1, random_agent)

# Pit the agents against eachother in the arena. Note that the player
# ids passed in need to match the index of the player in the array
arena = Arena(game, [player0, player1])
arena.play_games(verbose=True)
arena.statistics()
```

CHAPTER 2

Core

Core contains the primitives for playing a game between two players.

class core.**Player**(*player_id*, *agent*)

Player is a simple abstraction whose policy is defined by the agent that backs it. Agents learn the optimal play for each player, while players are only concerned about the optimal play for themselves

player_id

int – The id of the player

agent

Agent – The agent associated with the player

Raises ValueError – If the id is not 0 or 1

action(*g*, *s*, *flip*)

Take an action with the backing agent. If the starting player is not 0, then we invert the board so that the starting player is still 0 from the perspective of the agent

Parameters

- **g** (Game) – The game the player is playing
- **s** (*any*) – The state of the game
- **flip** (*bool*) – Whether or not to flip the state so that the agent thinks that player 0 started the game. This is necessary since trainable agents like MCTSAgent operate under the assumption that player 0 always starts

Returns The index of the action the player will take

Return type int

class core.**Arena**(*game*, *players*)

Place where two agents are pitted against eachother in a series of games. Statistics on the win rates are recorded and can be displayed.

game

Game – The game that is being played

players
list – List of Player objects. Note that there should only be two, and the ids of the player should map to the index of the player in the array.

games_played
int – The number of games played in the arena

wins
list – List of two integers representing the number of wins of each player, with the index being the id of the player

play_game(*verbose=False*)
Play a single game, doing the necessary bookkeeping to maintain accurate statistics and returning the winner (or -1 if no winner).

Note: We always have the start with player being 0 from the persepctive of the agent. Because of this we pass in a `flip` boolean to the player class in the action method, which flips the board and makes it seems as though player 0 started, even if it was actually player 1

Parameters **verbose** (*bool*) – Whether or not to print the output of the game. Defaults to false

Returns The winner of the game

Return type int

play_games(***kwargs*)
Play a series of games between the players, recording how they did so that we can display statistics on which player performed better

Parameters

- **num_episodes** (*int*) – The number of games to play, defaults to 10
- **verbose** (*bool*) – Whether or not to print output from each game. Defaults to false

statistics()
Print out the statistics for a given series of games.

class core.**Game**
Game class, which is extended to implement different types of adversarial, zero sum games. The class itself is stateless and all methods are actually static.

action_space(*s*)
For any given state returns a list of all possible valid actions

Parameters **s** (*any*) – The state of the game

flip_state(*s*)
Invert the state of the board so that player 0 becomes player 1

Parameters **s** (*any*) – The state of the game

initial_state()
Return the initial state of the game

next_state(*s*, *a*, *p*)
Given a state, action, and player id, return the state resulting from the player making that move

Parameters

- **s** (*any*) – The state of the game
- **a** (*int*) – The action for the player to take
- **p** (*int*) – The player to get the next state for

reward(*s*, *p*)
Returns the reward for a given state

Parameters

- **s** (*any*) – The state of the game
- **p** (*int*) – The player to get the reward for

terminal(*s*)
Returns whether a given state is terminal

Parameters **s** (*any*) – The state of the game

to_hash(*s*)
Returns a hash of the game state, which is necessary for some algorithms such as MCTS

Parameters **s** (*any*) – The state of the game

to_readable_string(*s*)
Returns a pretty-formatted representation of the board

Parameters **s** (*any*) – The state of the game

winner(*s*)
Returns the winner of a game, or -1 if there is no winner

Parameters **s** (*any*) – The state of the game

Note: There are two types of agents, agents that are trainable and agents that are not. If an agent is trainable then it inherits from the `TrainableAgent` class and must implement all of the members defined below. For example, `MCTSAgent` is a trainable agent, while `MinimaxAgent` is not.

class core.**Agent**
An agent class which exposes a method called action. Given a certain state of a game and the player that is playing, the agent retuns the best action it can find, given a certain heuristic or strategy

action(*g*, *s*, *p*)
Given a game, a state of the game, return an action

Parameters

- **g** (Game) – The game the agent is competing in
- **s** (*any*) – The state of the game
- **p** (*int*) – The current player (either 0 or 1)

Returns The index of the action within the returned action space

Return type int

class core.**TrainableAgent**
Class that extends the functionality of a normal agent. This is necessary because agents are bound to a particular player, but for some algorithms the agent is really being trained to play optimally for both plays, so we have this class house the training data and then pass it into the agents when they are instantiated to avoid duplicated work

train(*g*, ***kwargs*)
Train the agent. As a convenience this should return `self.training_params()` at the end of training

Parameters **g** (Game) – The game the agent is training on

Returns The training params of the agent

Return type tuple

train_episode(*g*, ***kwargs*)
Single training iteration

Parameters **g** (Game) – The game the agent is training on

training_params(*g*)
Return the params that result from training

Parameters **g** (Game) – The game the agent is training on

class core.**Algorithm**
A basic abstraction for a class that finds an action to take in a given state for a given player. Even if the algorithm is not stateful it is still implemented as a class to provide a uniform interface.

Note: Despite this interface being almost identical to an agent, agents can use multiple algorithms to come up with an action for a player to execute in a game.

---

best_action(*g*, *s*, *p*)
Return the best action given a state and player

Parameters

- **g** (Game) – The game object
- **s** (*any*) – The current state of the game
- **p** (*int*) – The current player

Returns The best action the algorithm can find

Return type int

CHAPTER 3

Agents

Agents are essentially actors that takes actions based on some heuristic, either randomly in the case of *RandomAgent* or based on some trained information, in the case of *MCTSAgent*.

class agents.**RandomAgent**
Implementation of a random agent, which simply selects a random action from the current action space each turn

class agents.**HumanAgent**
Human agent, which waits for human input to determine what action to take. Note that they should input an integer corresponding to the index of the action they want to select

class agents.**MinimaxAgent**(***kwargs*)
Implementation of minimax which allows you to specify a cutoff horizon for the search.

minimax
Minimax – Algorithm that runs the minimax search

class agents.**MCTSAgent**
Agent that uses Monte Carlo Tree Search (MCTS)

mcts
MTCS – The mcts search class

CHAPTER 4

Games

Games are the environments in which two players are pitted.

class games.**TicTacToe**

Implements a 3x3 game of tictactoe, with state represented as an array of length 9. Currently the implementation is somewhat brittle and cannot be extended to an nxn board easily.

Examples

```
>>> TicTacToe().initial_state()
[-1, -1, -1, -1, -1, -1, -1, -1, -1]
```

```
>>> TicTacToe().to_readable_string([-1, 1, -1, 0, 0, -1, -1, 1, -1])
   | O |
-----------
 X | X |
-----------
   | O |
```

class games.**LineTacToe**

Implements a 1x3 tictactoe-like, with state represented as an array of length 3. The goal of the game is to get two consecutive xs or os. For example, [o, o, x] is winning for o. Note that whoever starts the game should win, every time, as going in the center will win the game. However this is a good game to test new agent / algorithm implementations as the entire state space is only 11 states.

Examples

```
>>> LineTacToe().initial_state()
[-1, -1, -1]
```

CHAPTER 5

Algorithms

Collection of useful AI algorithms

class algorithms.**MCTS**

Implementation of a Monte Carlo Tree Search. We want to learn how to play a game by keeping track of the best action in any state. We will do this by propagating whether or not the current player won the game back up through the game history. After enough iterations of game simulations we can choose the best move based on this stored information

wins

dict – A dictionary where the key is a tuple `(player, state_hash)` and the value is the number of wins that occurred at that state for the player. Note that the player represents whoever *played* the move in the state.

plays

dict – A dictionary of the same format as wins which represents the number of times the player made a move in the given state

best_action(*g*, *s*, *p*)

Get the best action for a given player in a given game state

Parameters

- **g** (Game) – The game
- **s** (*state*) – The current state of the game
- **p** (*int*) – The current player

Returns The best action given the current knowledge of the game

Return type int

execute_episode(*g*, *nnet=None*, *c_punt=1.4*)

Execute a single iteration of the search and update the internal state based on the generated examples

Parameters

- **g** (Game) – The game

- **nnet** (*Network*) – Optional nework to be used for the policy instead of a naive random playout
- **c_punt** (*float*) – The degree of exploration. Defaults to 1.4

monte_carlo_action(*g*, *s*, *p*, *c_punt*)

Choose an action during self play based on the UCB1 algorithm. Instead of just choosing the action that led to the most wins in the past, we choose the action that balances this concern with exploration

Parameters

- **g** (Game) – The game
- **s** (*any*) – The state of the game
- **p** (*int*) – The player who is about to make a move
- **c_punt** (*float*) – The degree of exploration

Returns

Tuple (best_move, expand), where playout is a boolean denoting whether or not the expansion phase has begun

Return type tuple

pi(*g*, *s*)

Return the favorability of each action in a given state

Parameters

- **g** (Game) – The game
- **s** (*any*) – The state to evaluate

Returns The favorabiltiy of each action

Return type `list` of `float`

static random_playout(*g*, *s*, *p*, *max_moves=1000*)

Perform a random playout and return the winner

Parameters

- **g** (Game) – The game
- **s** (*any*) – The state of the game to start the playout from
- **p** (*player*) – The player whose turn it currently is
- **max_moves** (*int*) – Maximum number of moves before the function exits

Returns The winner of the game, or -1 if there is not one

Return type int

search(*g*, *num_iters=100*, *verbose=False*, *nnet=None*, *c_punt=1.4*)

Play out a certain number of games, each time updating our win and play counts for any state that we visit during the game. As we continue to play, num_wins / num_plays for a given state should begin to converge on the true optimality of a state

Parameters

- **g** (Game) – Game to train on
- **num_iters** (*int*) – Number of search iterations
- **verbose** (*bool*) – Whether or not to render a progress bar

- **nnet** (*Network*) – Optional nework to be used for the policy instead of a naive random playout
- **c_punt** (*float*) – The degree of exploration. Defaults to 1.4

search_episode(*g*, *nnet=None*, *c_punt=1.4*)
We play a game by starting in the boards starting state and then choosing a random move. We then move to the next state, keeping track of which moves we chose. At the end of the game we go through our visited list and update the values of wins and plays so that we have a better understanding of which states are good and which are bad

Parameters

- **g** (Game) – Game to search
- **nnet** (*Network*) – Optional nework to be used for the policy instead of a naive random playout
- **c_punt** (*float*) – The degree of exploration. Defaults to 1.4

Returns

List of examples where each entry is of the format `[player, state_hash, reward]`

Return type list

update(*examples*)
Backpropagate the result of the training episodes

Parameters **examples** (*list*) – List of examples where each entry is of the format `[player, state_hash, reward]`

class algorithms.**Minimax**(*horizon=inf*)
Implementation of the minimax algorithm.

horizon
int – The max depth of the search. Defaults to infinity. Note that if this is set then the game's hueristic is used

best_action(*g*, *s*, *p*)
Return the best action given a state and player

Parameters

- **g** (Game) – The game object
- **s** (*any*) – The current state of the game
- **p** (*int*) – The current player

Returns The best action the algorithm can find

Return type int

max_play(*g*, *s*, *p*, *depth*)
Get the largest value of all the child nodes

Parameters

- **g** (Game) – The game
- **s** (*any*) – The state of the game upon execution
- **p** (*int*) – The current player (who is about to make a move)
- **depth** (*int*) – The current depth of the search tree

Returns The largest value of all the child states

Return type int

min_play(*g*, *s*, *p*, *depth*)
Get the smallest value of all the child nodes

Parameters

- **g** (Game) – The game
- **s** (*any*) – The state of the game upon execution
- **p** (*int*) – The current player (who is about to make a move)
- **depth** (*int*) – The current depth of the search tree

Returns The smallest value of all the child states

Return type int

Index

U

W