Package: tictactoe (via r-universe)

September 18, 2024

Type Package

Title Tic-Tac-Toe Game

Version 0.2.3

Description Implements tic-tac-toe game to play on console, either with human or AI players. Various levels of AI players are trained through the Q-learning algorithm.

License MIT + file LICENSE

LazyData TRUE

RoxygenNote 6.0.1

Depends R (>= 2.10)

Imports hash, stats

Suggests testthat, combiter,

URL https://github.com/kota7/tictactoe

BugReports https://github.com/kota7/tictactoe/issues

Repository https://kota7.r-universe.dev

RemoteUrl https://github.com/kota7/tictactoe

RemoteRef HEAD

RemoteSha 638244ec8fe832bb3af0fccbd083b0a0ad52094c

Contents

`equivalent_states` *Equivalent States*

Description

Returns a set of equivalent states and actions

Usage

```
equivalent_states(state)

equivalent_states_actions(state, action)
```

Arguments

`state`	state, 3x3 matrix
`action`	integer vector of indices (1 to 9)

Value

`equivalent_states` returns a list of state matrices

`equivalent_states_actions` returns a list of two lists: `states`, the set of equivalent states and `actions`, the set of equivalent actions

---

`hash-ops` *Hash Operations for Single State*

Description

Hash Operations for Single State

Usage

```
haskey(x, ...)

## S3 method for class 'xhash'
x[state, ...]

## S3 replacement method for class 'xhash'
x[state, ...] <- value

## S3 method for class 'xhash'
haskey(x, state, ...)
```

Arguments

x	object
...	additional arguments to determine the key
state	state object
value	value to assign

Value

- haskey returns a logical
- `[` returns a reference to the object
- `[<-` returns a value

---

ttt *Play Tic-Tac-Toe Game*

Description

Start tic-tac-toe game on the console.

Usage

```
ttt(player1 = ttt_human(), player2 = ttt_human(), sleep = 0.5)
```

Arguments

player1, player2	objects that inherit ttt_player class
sleep	interval to take before an AI player to make decision, in second

Details

At default, the game is played between humans. Set player1 or player2 to ttt_ai() to play against an AI player. The strength of the AI can be adjusted by passing the level argument (0 (weekest) to 5 (strongest)) to the ttt_ai function.

To input your move, type the position like "a1". Only two-length string consisting of an alphabet and a digit is accepted. Type "exit" to finish the game.

You may set both player1 and player2 as AI players. In this case, the game transition is displayed on the console without human inputs. For conducting a large sized simulations of games between AIs, refer to ttt_simulate

See Also

ttt_ai, ttt_human, ttt_simulate

Examples

```
## Not run:
ttt(ttt_human(), ttt_random())

## End(Not run)
```

`ttt_ai` *Tic-Tac-Toe AI Player*

Description

Create an AI tic-tac-toe game player

Usage

```
ttt_ai(name = "ttt AI", level = 0L)

ttt_random(name = "random AI")
```

Arguments

`name`	player name
`level`	AI strength. must be Integer 0 (weekest) to 5 (strongest)

Details

`level` argument controls the strength of AI, from 0 (weekest) to 5 (strongest). `ttt_random` is an alias of `ttt_ai(level = 0)`.

A `ttt_ai` object has the `getmove` function, which takes `ttt_game` object and returns a move considered as optimal. `getmove` function is designed to take a `ttt_game` object and returns a move using the policy function.

The object has the value and policy functions. The value function maps a game state to the evaluation from the first player's viewpoint. The policy function maps a game state to a set of optimal moves in light of the value evaluation. The functions have been trained through the Q-learning.

Value

`ttt_ai` object

Fields

`name` Player name

`level` Strength (0 to 5)

`policy_func` `xhash` object that maps a game state to moves

`value_func` `xhash` object that maps a game state to a value

Methods

`getmove(game, ...)` Returns a move considered as optimal.

Input:

- `game`: `ttt_game` object

Output: a move

Examples

```
game <- ttt_game()
p <- ttt_ai(level=3)
p$getmove(game)
```

`ttt_game` *Tic-Tac-Toe Game*

Description

Object that encapsulates a tic-tac-toe game.

Usage

```
ttt_game()
```

Value

`ttt_game` object

Fields

`state` 3 x 3 matrix of current state

`nextmover`, `prevmover` Next and previous mover (1 or 2)

`history` N x 2 matrix of game history, each row represents a move by (player, position)

Methods

`play(position, ...)` Play a move. At default, play is made by the next mover, but can be changed by setting the 'nextmover' argument.

Input:

- `position`: position to play
- `...`: Variables to overload

Output: `TRUE` iff a move is legal and game has not been over.

`undo()` Undo the previous play

Input: None

Output: `NULL`

`is_legal(position)` Check if the position is a legal move

Input:

- `position`: position to check

Output: TRUE if the given position is a legal move

`legal_moves()` Returns all legal moves

Input: None

Output: Integer vector of legal moves

`check_win(player)` Check if the given player has won.

Input:

- `player`: player (1 or 2)
- `...`: Variables to be overloaded

Output: TRUE iff the given player has won

`check_result()` Check the result from the board state

Input: None

Output:

- `-1`: undetermined yet
- `0`: draw
- `1`: won by player 1
- `2`: won by player 2

`next_state(position, ...)` Returns the hypothetical next state without changing the `state` field.

Input:

- `position`: position to play

Output: `state` matrix

`show_board()` print the boad on consle

Input: None

Output: NULL

`to_index(position)` Convert a position to the index

Input:

- `position`: a position

Output: an integer 1 to 9, or 0 for a invalid position

`index_to_str(position)` Convert a position to a location representation in the form of "A1"

Input:

- `position`: a position

Output: a character

Examples

```
x <- ttt_game()
x$play(3)
x$play(5)
x$show_board()

x$undo()
x$show_board()
```

`ttt_human` *Human Tic-Tac-Toe Player*

Description

Create an human tic-tac-toe player

Usage

```
ttt_human(name = "no name")
```

Arguments

`name`	player name

Value

`ttt_human` object

Fields

`name` Player name

Methods

`getmove(game, prompt = "choose move (e.g. A1) > ", ...)` Communicate with users to type in the next move.

Input:

- `game`: `ttt_game` object
- `prompt`: prompt message

Output: a character of a move

Examples

```
## Not run:
p <- ttt_human()
p$getmove()

## End(Not run)
```

`ttt_qlearn` *Q-Learning for Training Tic-Tac-Toe AI*

Description

Train a tic-tac-toe AI through Q-learning

Usage

```
ttt_qlearn(player, N = 1000L, epsilon = 0.1, alpha = 0.8, gamma = 0.99,
  simulate = TRUE, sim_every = 250L, N_sim = 1000L, verbose = TRUE)
```

Arguments

`player`	AI player to train
`N`	number of episode, i.e. training games
`epsilon`	fraction of random exploration move
`alpha`	learning rate
`gamma`	discount factor
`simulate`	if true, conduct simulation during training
`sim_every`	conduct simulation after this many training games
`N_sim`	number of simulation games
`verbose`	if true, progress report is shown

Details

This function implements Q-learning to train a tic-tac-toe AI player. It is designed to train one AI player, which plays against itself to update its value and policy functions.

The employed algorithm is Q-learning with epsilon greedy. For each state s, the player updates its value evaluation by

$$V(s) = (1 - \alpha)V(s) + \alpha\gamma max'_s V(s')$$

if it is the first player's turn. If it is the other player's turn, replace max by min. Note that s' spans all possible states you can reach from s. The policy function is also updated analogously, that is, the set of actions to reach s' that maximizes $V(s')$. The parameter α controls the learning rate, and $gamma$ is the discount factor (earlier win is better than later).

Then the player chooses the next action by ϵ-greedy method; Follow its policy with probability $1 - \epsilon$, and choose random action with probability ϵ. ϵ controls the ratio of explorative moves.

At the end of a game, the player sets the value of the final state either to 100 (if the first player wins), -100 (if the second player wins), or 0 (if draw).

This learning process is repeated for `N` training games. When `simulate` is set true, simulation is conducted after `sim_every` training games. This would be usefule for observing the progress of training. In general, as the AI gets smarter, the game tends to result in draw more.

See Sutton and Barto (1998) for more about the Q-learning.

Value

data.frame of simulation outcomes, if any

References

Sutton, Richard S and Barto, Andrew G. Reinforcement Learning: An Introduction. The MIT Press (1998)

Examples

```
p <- ttt_ai()
o <- ttt_qlearn(p, N = 200)
```

---

ttt_simulate *Simulate Tic-Tac-Toe Games between AIs*

Description

Simulate Tic-Tac-Toe Games between AIs

Usage

```
ttt_simulate(player1, player2 = player1, N = 1000L, verbose = TRUE,
  showboard = FALSE, pauseif = integer(0))
```

Arguments

player1, player2	AI players to simulate
N	number of simulation games
verbose	if true, show progress report
showboard	if true, game transition is displayed
pauseif	pause the simulation when specified results occur. This can be useful for explorative purposes.

Value

integer vector of simulation outcomes

Examples

```
res <- ttt_simulate(ttt_ai(), ttt_ai())
prop.table(table(res))
```

vectorized-hash-ops *Vectorized Hash Operations*

Description

Vectorized Hash Operations

Usage

```
haskeys(x, ...)

setvalues(x, ...)

getvalues(x, ...)

## S3 method for class 'xhash'
getvalues(x, states, ...)

## S3 method for class 'xhash'
setvalues(x, states, values, ...)

## S3 method for class 'xhash'
haskeys(x, states, ...)
```

Arguments

x	object
...	additional arugments to determine the keys
states	state object
values	values to assign

Value

- haskeys returns a logical vector
- setvalues returns a reference to the object
- getvalues returns a list of values

xhash *Create Hash Table for Generic States*

Description

This function creates an xhash object, extended version of hash. While hash accepts only strings as indices, xhash can deal with generic index variables, termed as "state".

Usage

```
xhash(convfunc = function(state, ...) state, convfunc_vec = function(states,
  ...) unlist(Map(convfunc, states, ...)), default_value = NULL)
```

Arguments

convfunc	function that converts a state to a key. It must take a positional argument state and keyword arguments represented by ..., and returns a character.
convfunc_vec	function for vectorized conversion from states to keys. This function must receive a positional argument states and keyword arguments ... and returns character vector. By default, it vectorizes convfunc using Map. User may specify a more efficient function if any.
default_value	value to be returned when a state is not recorded in the table.

Value

xhash object

See Also

hash-ops, vectorized-hash-ops

Examples

```
h <- xhash(convfunc = function(state, ...) paste0(state, collapse='-'))

# insert
h[c(1, 2, 3)] <- 100
h[matrix(1:9, nrow=3, ncol=3)] <- -5

# retrieve
h[c(1, 2, 3)]
h[matrix(1:9, nrow=3, ncol=3)]
h[1:9]          # equivalent as above, due to conversion to a same key
h[c(3, 2, 1)]   # this is undefined

# delete
h[c(1, 2, 3)] <- NULL
```

```
# vectorized operations
## insert
setvalues(h, list(1:2, 1:3), c(9, 8))
## retrieve
getvalues(h, list(1:9, 1:2, 3:1))
## delete
setvalues(h, list(1:9, 1:3), NULL)
```

Index