BlackBird Documentation

Release July 2018

Jordan Singer, Kevin Welsh

Jul 14, 2018

Contents:

BlackBird is a Python implementation of the AlphaZero algorithm. It contains base classes for running Monte Carlo Tree Search (MCTS), neural networks running on the TensorFlow framework, and a template for loading in custom board games for the system to self-learn.

CHAPTER 1

BlackBird Suite

1.1 Blackbird module

class Blackbird.**ExampleState**(*evaluation*, *policy*, *board*, *player=None*)
Bases: object

Class which centralizes the data structure of a game state.

GameState objects have many properties, but only a few of them are relevant to training. *ExampleState* provides an interface on top of protocol buffers for reading and storing game state data.

`MctsPolicy`
A numpy array which holds the policy generated from applying MCTS.

`MctsEval`
A float between -1 and 1 representing the evaluation that the MCTS computed.

`Board`
A numpy array which holds the input state for a board state. In general, this is the game state, as well as layers for historical positions, current turn, current player, etc.

`Player`
An optional integer, representing the current player.

classmethod FromSerialized(*serialState*)
Transforms a protobuf bytecode string to an *ExampleState* object.

Parameters **serialState** – A protobuf bytecode string holding *GameState* info.

Returns An *ExampleState* object holding the relevant deserialized data.

SerializeState()
Returns the protobuf bytecode serialization of the *ExampleState*.

Blackbird.**GenerateTrainingSamples**(*model*, *nGames*, *temp*)
Generates self-play games to learn from.

This method generates *nGames* self-play games, and stores the game states in a local sqlite3 database.

Parameters

- **model** – The Blackbird model to use to generate games
- **nGames** – An int determining the number of games to generate.
- **temp** – A float between 0 and 1 determining the exploration temp for MCTS. Usually this should be close to 1 to ensure high move exploration rate.

Raises `ValueError` – nGames was not a positive integer.

class `Blackbird.`**Model**(*game*, *name*, *mctsConfig*, *networkConfig={}*, *tensorflowConfig={}*)

Bases: *DynamicMCTS.DynamicMCTS*, *Network.Network*

Class which encapsulates MCTS powered by a neural network.

The BlackBird class is designed to learn how to win at a board game, by using Monte Carlo Tree Search (MCTS) with the tree search powered by a neural network. :param *game*: A GameState object which holds the rules of the game

BlackBird is intended to learn.

Parameters

- **name** – The name of the model.
- **mctsConfig** – JSON config for MCTS runtime evaluation
- **networkConfig** – JSON config for creating a new network from NetworkFactory
- **tensorflowConfig** – Configuaration for tensorflow initialization

GetPriors

Returns BlackBird's policy of a supplied position.

BlackBird's network will evaluate the policy of a supplied position.

Parameters **state** – A GameState object which should be evaluated.

Returns

A list of floats of size *len(state.LegalActions())* which sums to 1, representing the probabilities of selecting each legal action.

Return type *policy*

LastVersion()

SampleValue

Returns BlackBird's evaluation of a supplied position.

BlackBird's network will evaluate a supplied position, from the perspective of *player*.

Parameters

- **state** – A GameState object which should be evaluated.
- **player** – An int representing the current player.

Returns

A float between 0 and 1 holding the evaluation of the position. 0 is the worst possible evaluation, 1 is the best.

Return type *value*

Blackbird.**TestGood**(*model*, *temp*, *numTests*)
Plays the current BlackBird instance against a standard MCTS player.

Game statistics are logged in the local *data/blackbird.db* database.

Parameters

- **model** – The Blackbird model to test
- **temp** – A float between 0 and 1 determining the exploitation temp for MCTS. Usually this should be close to 0.1 to ensure optimal move selection.
- **numTests** – An int determining the number of games to play.

Returns

- *wins*: The number of wins *model* had.
- *draws*: The number of draws *model* had.
- *losses*: The number of losses *model* had.

Return type A dictionary holding

Blackbird.**TestModels**(*model1*, *model2*, *temp*, *numTests*)
Base function for playing a BlackBird instance against another model.

Parameters

- **model1** – The Blackbird model to test.
- **model2** – The model to play against.
- **temp** – A float between 0 and 1 determining the exploitation temp for MCTS. Usually this should be close to 0.1 to ensure optimal move selection.
- **numTests** – An int determining the number of games to play.

Returns An integer representing a win (1), draw (0), or loss (-1)

Blackbird.**TestPrevious**(*model*, *temp*, *numTests*)
Plays the current BlackBird instance against the previous version of BlackBird's neural network.

Game statistics are logged in the local *data/blackbird.db* database.

Parameters

- **model** – The Blackbird model to test
- **temp** – A float between 0 and 1 determining the exploitation temp for MCTS. Usually this should be close to 0.1 to ensure optimal move selection.
- **numTests** – An int determining the number of games to play.

Returns

- *wins*: The number of wins *model* had.
- *draws*: The number of draws *model* had.
- *losses*: The number of losses *model* had.

Return type A dictionary holding

Blackbird.**TestRandom**(*model*, *temp*, *numTests*)
Plays the current BlackBird instance against an opponent making random moves.

Game statistics are logged in the local *data/blackbird.db* database.

Parameters

- **temp** – A float between 0 and 1 determining the exploitation temp for MCTS. Usually this should be close to 0.1 to ensure optimal move selection.
- **numTests** – An int determining the number of games to play.

Returns

- *wins*: The number of wins *model* had.
- *draws*: The number of draws *model* had.
- *losses*: The number of losses *model* had.

Return type A dictionary holding

Blackbird.**TrainWithExamples**(*model*, *batchSize*, *learningRate*, *teacher=None*)

Trains the neural network on provided example positions.

Provided a list of example positions, this method will train BlackBird's neural network to play better. If *teacher* is provided, the neural network will include a cross-entropy term in the loss calculation so that the other network's policy is incorporated into the learning.

Parameters

- **model** – The Blackbird model to train
- **examples** – A list of *TrainingExample* objects which the neural network will learn from.
- **teacher** – An optional *BlackBird* object whose policy the current network will include in its loss calculation.

1.2 Connect4 module

class Connect4.**BoardState**

Bases: *GameState.GameState*

ApplyAction(*action*)

AsInputArray()

BoardShape = <MagicMock name='mock()' id='139788815285944'>

Copy()

Dirs = [(0, 1), (1, 1), (1, 0), (1, -1)]

EvalToString(*eval*)

GameType = 'Connect4'

Height = 6

InARow = 4

LegalActionShape()

LegalActions()

LegalMoves = 7

Players = {0: ' ', 1: 'X', 2: 'O'}

Width = 7

Winner(*prevAction=None*)

1.3 DynamicMCTS module

class DynamicMCTS.**DynamicMCTS**(***kwargs*)
Bases: *MCTS.MCTS*

An extension of the MCTS class that aggregates statistics as it explores.

This class is identical to base MCTS, with the class function FindLeaf defined.

_findLeaf(*node*, *temp*)
Applies MCTS to a supplied node until a leaf is found.

Parameters

- **node** – A Node object to find a leaf of.
- **temp** – The temperature to apply to action selection after tree search has been applied to the node.

1.4 FixedMCTS module

class FixedMCTS.**FixedMCTS**(***kwargs*)
Bases: *MCTS.MCTS*

An implementation of Monte Carlo Tree Search that only aggregates statistics up to a fixed depth.

1.5 GameState module

class GameState.**GameState**
Bases: object

ApplyAction(*action*)

Copy()

EvalToString(*eval*)

LegalActionShape()

LegalActions()

NumericRepresentation()

SerializeState(*state*, *policy*, *eval*)

Winner(*prevAction=None*)

1.6 MCTS module

class MCTS.**MCTS**(*explorationRate*, *timeLimit=None*, *playLimit=None*, ***kwargs*)
Bases: object

Base class for Monte Carlo Tree Search algorithms.

Outlines all the necessary operations for the core MCTS algorithm. _findLeaf() will need to be overriden to avoid a NotImplemenetedError.

TimeLimit
The default max move time in seconds.

PlayLimit
The default number of positions to evaluate per move.

ExplorationRate
The exploration parameter for MCTS.

Root
The Node object representing the root of the MCTS.

AddChildren(*node*)
Expands a node and adds children, actions and priors.

Given a node, MCTS will evaluate the node's children, if they exist. The evaluation and prior policy are supplied in the creation of the child Node object.

Parameters **node** – A Node object to expand.

DropRoot()
Resets self.Root to None

FindMove(*state*, *temp=0.1*, *moveTime=None*, *playLimit=None*)
Finds the optimal move in a position.

Given a game state, this will use a Monte Carlo Tree Search algorithm to pick the best next move.

Parameters

- **state** – A GameState object which the function will evaluate.
- **temp** – A float determining the temperature to apply in move selection.
- **moveTime** – An optional float determining the allowed search time.
- **playLimit** – An optional float determining the allowed number of positions to evaluate.

Returns

A tuple providing, in order...

- The board state after applying the selected move
- The decided value of input state
- The probabilities of choosing each of the children

Raises

- `TypeError` – state was not an object of type GameState.
- `ValueError` – The function was not able to determine a stop time.

GetPriors(*state*)
Gets the array of prior search probabilities.

This is the default GetPriors for MCTS. The return value is always an array of ones. This should be overridden to get actual utility.

Parameters **state** – A GameState object to get the priors of.

Returns A numpy array of ones of shape [num_legal_actions_of_state].

MoveRoot(*state*)
This is the public API of MCTS._moveRoot.

Move the root of the tree to the provided state. Use this to update the root so that tree integrity can be maintained between moves if necessary. Does nothing if Root is None, for example after running DropRoot().

Parameters **state** – A GameState object which self.Root should be updated to.

ResetRoot()
Set self.Root to the appropriate initial state.

Reset the state of self.Root to an appropriate initial state. If self.Root was already None, then there is nothing to do, and it will remain None. Otherwise, ResetRoot will apply an iterative backup to self.Root until its parent is None.

SampleValue(*state*, *player*)
Samples the value of a state for a specified player.

This applies a set of Monte Carlo random rollouts to a state until a game terminates, and returns the determined evaluation.

Parameters

- **state** – A GameState object which the function will obtain the evaluation of.
- **player** – An integer representing the current player in state.

Returns

A float representing the value of the state. It is 0 if it was determined to be a loss, 1 if it was determined to be a win, and 0.5 if it was determined to be a draw.

_applyAction(*state*, *action*)
Applies an action to a provided state.

Parameters

- **state** – A GameState object which needs to be updated.
- **action** – An int which indicates the action to apply to state.

_backProp(*leaf*, *stateValue*, *playerForValue*)
Backs up a value from a leaf through to self.Root.

Given a leaf node and a value, this function will back-propogate the value to its parent node, and propogate that all the way through the tree to its root, self.Root

Parameters

- **leaf** – A Node object which is the leaf of the current tree to apply back-propogation to.
- **stateValue** – The MCTS-created evaluation to back-propogate.
- **playerForValue** – The player which stateValue applies to.

_findLeaf(*node*, *temp*)
Applies MCTS to a supplied node until a leaf is found.

Parameters **node** – A Node object to find a leaf of.

_moveRoot(*state*)
Updates the root of the tree.

Move the root of the tree to the provided state. Use this to update the root so that tree integrity can be maintained between moves if necessary. Does nothing if Root is None, for example after running DropRoot().

Parameters **state** – A GameState object which self.Root should be updated to.

_runMCTS(*temp*, *endTime=None*, *nPlays=None*)
Run the MCTS algorithm on the current Root Node.

Given the current game state, represented by self.Root, a child node is seleted using the _findLeaf method. This method will apply temp to all child node move selection proportions, compute the sampled value of the action, and backpropogate the value through the tree.

Parameters

- **temp** – A float determining the temperature to apply in FindMove.
- **endTime** – (optional) The maximum time to spend on searching.
- **nPlays** – (optional) The maximum number of positions to evaluate.

_selectAction(*root*, *temp*, *exploring=True*)
Chooses an action from an explored root.

Selects a child of the root using an upper confidence interval. If you are not exploring, setting the exploring flag to false will instead choose the one with the highest expected payout - ignoring the exploration/regret factor.

Parameters

- **root** – A Node object which must have children Nodes.
- **temp** – The temperature to apply to the children Node visit counts. If temp is 0, _selectAction will return the child Node with the greatest visit count.
- **exploring** – A boolean toggle for overriding the selection type to a simple argmax. If True, _selectAction will return the child Node with the greatest visit count.

Returns An int representing the index of the selected action.

Return type choice

class MCTS.**Node**(*state*, *legalActions*, *priors*, ***kwargs*)
Bases: object

Base class for storing game state information in tree searches.

This is the abtract tree node class that is used to cache/organize game information during the search.

State
A GameState object holding the Node's state representation.

Value
A float holding the Node's state valuation.

Plays
A counter holding the number of times the Node has been used.

LegalActions
An int holding the number of legal actions for the Node.

Children
A list of Nodes holding all legal states for the Node.

Parent
A Node object representing the Node's parent.

Priors
A numpy array of size [num_legal_actions] that holds the Node's prior probabilities. At instantiation, the provided prior is filtered on only legal moves.

_childWinRates
A numpy array of size [num_legal_actions] used for storing the win rates of the Node's children in MCTS.

_childPlays
A numpy array of size [num_legal_actions] used for storing the play counts of the Node's children in MCTS.

ChildPlays()
Samples the play rate of each child Node object.

Samples the play rates for each of the Node's children. Not helpful if none of the children have been evaluated in MCTS.

Returns A numpy array representing the play rate for each of the Node's children.

ChildProbability()
Samples the probabilities of sampling each child Node.

Samples the play rate for each of the Node's children Node objects. If no children have been sampled in MCTS, this returns zeros.

Returns A numpy array representing the play rate for each of the Node's children. Defaults to an array of zeros if no children have been sampled.

ChildWinRates()
Samples the win rate of each child Node object.

Samples the win rates for each of the Node's children. Not helpful if none of the children have been evaluated in MCTS.

Returns A numpy array representing the win rate for each of the Node's children.

WinRate()
Samples the win rate of the Node after MCTS.

This is a simple API which provides the win rate of the Node after applying MCTS.

Returns A float representing the win rate for the Node. If no plays have been applied to this Node, the default value of 0 is returned.

1.7 Network module

class Network.**Network**(*name*, *networkConstructor=None*, *tensorflowConfig={}*)
Bases: object

getEvaluation(*state*)
Given a game state, return the network's evaluation.

getPolicy(*state*)
Given a game state, return the network's policy. Random Dirichlet noise is applied to the policy output to ensure exploration, if training.

grabVariables()

loadModel(*name*)
Load an old version of the network.

saveModel(*name=None*)
Write the state of the network to a file. This should be reserved for "best" networks.

train(*state*, *eval*, *policy*, *learningRate=0.01*, *teacher=None*)
Train the network

1.8 RandomMCTS module

class RandomMCTS.**RandomMCTS**(**args*, ***kwargs*)
Bases: *MCTS.MCTS*

FindMove(*state*, **args*, ***kwargs*)
Finds the optimal move in a position.

Given a game state, this will use a Monte Carlo Tree Search algorithm to pick the best next move.

Parameters

- **state** – A GameState object which the function will evaluate.
- **temp** – A float determining the temperature to apply in move selection.
- **moveTime** – An optional float determining the allowed search time.
- **playLimit** – An optional float determining the allowed number of positions to evaluate.

Returns

A tuple providing, in order...

- The board state after applying the selected move
- The decided value of input state
- The probabilities of choosing each of the children

Raises

- `TypeError` – state was not an object of type GameState.
- `ValueError` – The function was not able to determine a stop time.

MoveRoot(**args*, ***kwargs*)
This is the public API of MCTS._moveRoot.

Move the root of the tree to the provided state. Use this to update the root so that tree integrity can be maintained between moves if necessary. Does nothing if Root is None, for example after running DropRoot().

Parameters **state** – A GameState object which self.Root should be updated to.

ResetRoot(**args*, ***kwargs*)
Set self.Root to the appropriate initial state.

Reset the state of self.Root to an appropriate initial state. If self.Root was already None, then there is nothing to do, and it will remain None. Otherwise, ResetRoot will apply an iterative backup to self.Root until its parent is None.

1.9 TicTacToe module

class TicTacToe.**BoardState**

Bases: *GameState.GameState*

ApplyAction(*action*)

AsInputArray()

BoardShape = <MagicMock name='mock()' id='139788795127400'>

Copy()

Dirs = [(0, 1), (1, 1), (1, 0), (1, -1)]

EvalToString(*eval*)

GameType = 'TicTacToe'

InARow = 3

LegalActionShape()

LegalActions()

LegalMoves = 9

Players = {0: ' ', 1: 'X', 2: 'O'}

Size = 3

Winner(*prevAction=None*)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

P

R

S

T

V

W