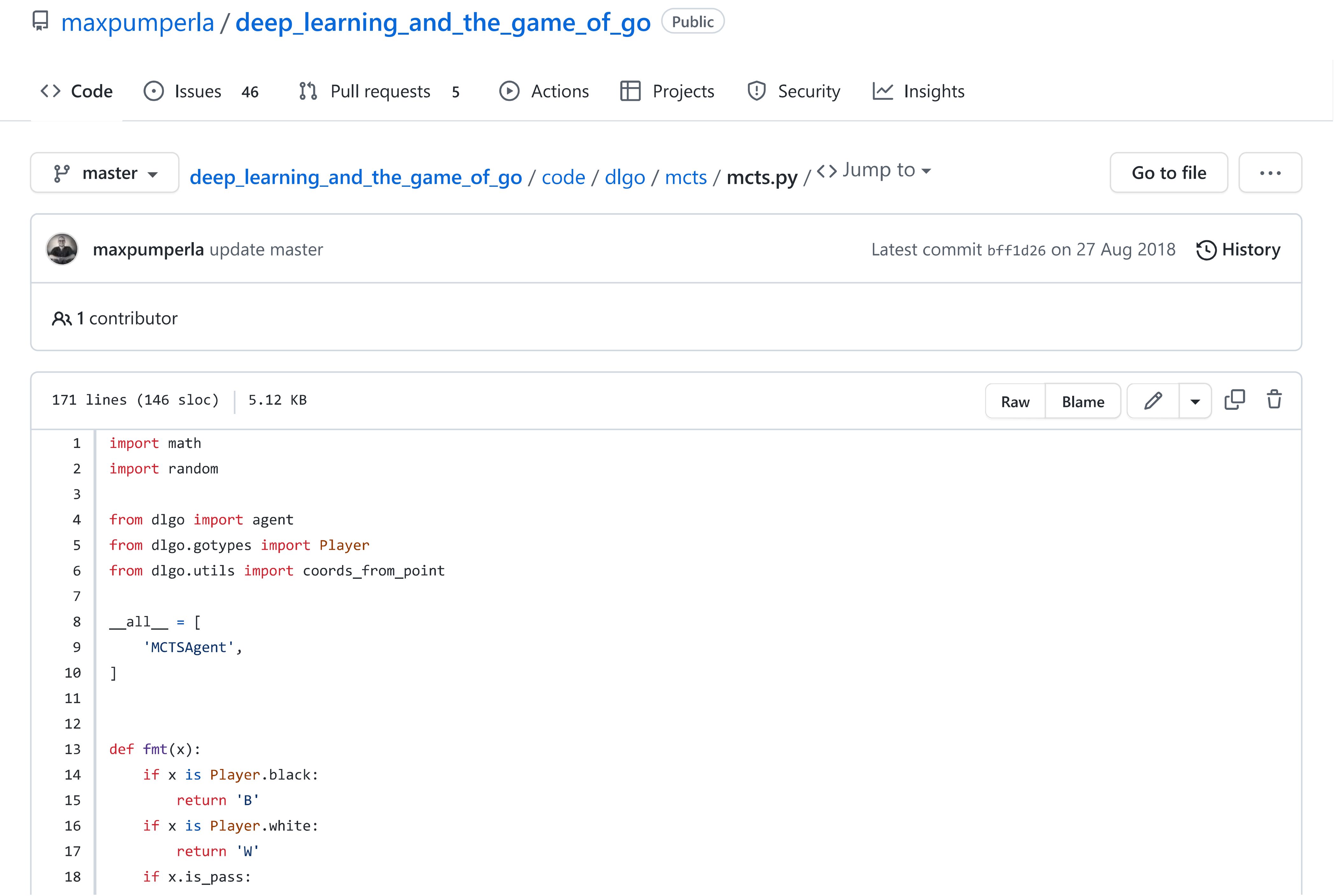
maxpumperla / deep_learning_and_the_game_of_go
Public
Code
Issues 46
Pull requests 5
Actions
Projects
Security
Insights
master
deep_learning_and_the_game_of_go / code / dlgo / mcts / mcts.py /
Jump to
Go to file
History
Raw
Blame

maxpumperla update master

Latest commit bff1d26 on 27 Aug 2018

1 contributor

171 lines (146 sloc) | 5.12 KB

```python
import math
import random

from dlgo import agent
from dlgo.gotypes import Player
from dlgo.utils import coords_from_point

__all__ = [
    'MCTSAgent',
]


def fmt(x):
    if x is Player.black:
        return 'B'
    if x is Player.white:
        return 'W'
    if x.is_pass:
        return 'pass'
    if x.is_resign:
        return 'resign'
    return coords_from_point(x.point)


def show_tree(node, indent='', max_depth=3):
    if max_depth < 0:
        return
    if node is None:
        return
    if node.parent is None:
        print('%sroot' % indent)
    else:
        player = node.parent.game_state.next_player
        move = node.move
        print('%s%s %s %d %.3f' % (
            indent, fmt(player), fmt(move),
            node.num_rollouts,
            node.winning_frac(player),
        ))
    for child in sorted(node.children, key=lambda n: n.num_rollouts, reverse=True):
        show_tree(child, indent + '  ', max_depth - 1)


# tag::mcts-node[]
class MCTSNode(object):
    def __init__(self, game_state, parent=None, move=None):
        self.game_state = game_state
        self.parent = parent
        self.move = move
        self.win_counts = {
            Player.black: 0,
            Player.white: 0,
        }
```

```python
        self.num_rollouts = 0
        self.children = []
        self.unvisited_moves = game_state.legal_moves()
# end::mcts-node[]

# tag::mcts-add-child[]
    def add_random_child(self):
        index = random.randint(0, len(self.unvisited_moves) - 1)
        new_move = self.unvisited_moves.pop(index)
        new_game_state = self.game_state.apply_move(new_move)
        new_node = MCTSNode(new_game_state, self, new_move)
        self.children.append(new_node)
        return new_node
# end::mcts-add-child[]

# tag::mcts-record-win[]
    def record_win(self, winner):
        self.win_counts[winner] += 1
        self.num_rollouts += 1
# end::mcts-record-win[]

# tag::mcts-readers[]
    def can_add_child(self):
        return len(self.unvisited_moves) > 0

    def is_terminal(self):
        return self.game_state.is_over()

    def winning_frac(self, player):
        return float(self.win_counts[player]) / float(self.num_rollouts)
# end::mcts-readers[]


class MCTSAgent(agent.Agent):
    def __init__(self, num_rounds, temperature):
        agent.Agent.__init__(self)
        self.num_rounds = num_rounds
        self.temperature = temperature

# tag::mcts-signature[]
    def select_move(self, game_state):
        root = MCTSNode(game_state)
# end::mcts-signature[]

# tag::mcts-rounds[]
        for i in range(self.num_rounds):
            node = root
            while (not node.can_add_child()) and (not node.is_terminal()):
                node = self.select_child(node)

            # Add a new child node into the tree.
            if node.can_add_child():
                node = node.add_random_child()

            # Simulate a random game from this node.
            winner = self.simulate_random_game(node.game_state)

            # Propagate scores back up the tree.
            while node is not None:
                node.record_win(winner)
                node = node.parent
# end::mcts-rounds[]

        scored_moves = [
            (child.winning_frac(game_state.next_player), child.move, child.num_rollouts)
            for child in root.children
        ]
        scored_moves.sort(key=lambda x: x[0], reverse=True)
        for s, m, n in scored_moves[:10]:
            print('%s - %.3f (%d)' % (m, s, n))
```

```python

# tag::mcts-selection[]
        # Having performed as many MCTS rounds as we have time for, we
        # now pick a move.
        best_move = None
        best_pct = -1.0
        for child in root.children:
            child_pct = child.winning_frac(game_state.next_player)
            if child_pct > best_pct:
                best_pct = child_pct
                best_move = child.move
        print('Select move %s with win pct %.3f' % (best_move, best_pct))
        return best_move
# end::mcts-selection[]

# tag::mcts-uct[]
    def select_child(self, node):
        """Select a child according to the upper confidence bound for
        trees (UCT) metric.
        """
        total_rollouts = sum(child.num_rollouts for child in node.children)
        log_rollouts = math.log(total_rollouts)

        best_score = -1
        best_child = None
        # Loop over each child.
        for child in node.children:
            # Calculate the UCT score.
            win_percentage = child.winning_frac(node.game_state.next_player)
            exploration_factor = math.sqrt(log_rollouts / child.num_rollouts)
            uct_score = win_percentage + self.temperature * exploration_factor
            # Check if this is the largest we've seen so far.
            if uct_score > best_score:
                best_score = uct_score
                best_child = child
        return best_child
# end::mcts-uct[]

    @staticmethod
    def simulate_random_game(game):
        bots = {
            Player.black: agent.FastRandomBot(),
            Player.white: agent.FastRandomBot(),
        }
        while not game.is_over():
            bot_move = bots[game.next_player].select_move(game)
            game = game.apply_move(bot_move)
        return game.winner()
```