```
# Copyright 2018 Google LLC
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""Monte Carlo Tree Search implementation.

All terminology here (Q, U, N, p_UCT) uses the same notation as in the
AlphaGo (AG) paper.
"""

import collections
import math

from absl import flags
import numpy as np

import coords
import go

# 722 moves for 19x19, 162 for 9x9
flags.DEFINE_integer('max_game_length', int(go.N ** 2 * 2),
                     'Move number at which game is forcibly terminated')

flags.DEFINE_float('c_puct_base', 19652,
                   'Exploration constants balancing priors vs. value net output.')

flags.DEFINE_float('c_puct_init', 1.25,
                   'Exploration constants balancing priors vs. value net output.')

flags.DEFINE_float('dirichlet_noise_alpha', 0.03 * 361 / (go.N ** 2),
                   'Concentrated-ness of the noise being injected into priors.')
flags.register_validator('dirichlet_noise_alpha', lambda x: 0 <= x < 1)

flags.DEFINE_float('dirichlet_noise_weight', 0.25,
                   'How much to weight the priors vs. dirichlet noise when mixing')
flags.register_validator('dirichlet_noise_weight', lambda x: 0 <= x < 1)

FLAGS = flags.FLAGS


class DummyNode(object):
    """A fake node of a MCTS search tree.

    This node is intended to be a placeholder for the root node, which would
    otherwise have no parent node. If all nodes have parents, code becomes
    simpler."""

    def __init__(self):
        self.parent = None
        self.child_N = collections.defaultdict(float)
        self.child_W = collections.defaultdict(float)


class MCTSNode(object):
    """A node of a MCTS search tree.
```

```
    A node knows how to compute the action scores of all of its children,
    so that a decision can be made about which move to explore next. Upon
    selecting a move, the children dictionary is updated with a new node.

    position: A go.Position instance
    fmove: A move (coordinate) that led to this position, a flattened coord
            (raw number between 0-N^2, with None a pass)
    parent: A parent MCTSNode.
    """

    def __init__(self, position, fmove=None, parent=None):
        if parent is None:
            parent = DummyNode()
        self.parent = parent
        self.fmove = fmove  # move that led to this position, as flattened coords
        self.position = position
        self.is_expanded = False
        self.losses_applied = 0  # number of virtual losses on this node
        # using child_() allows vectorized computation of action score.
        self.illegal_moves = 1 - self.position.all_legal_moves()
        self.child_N = np.zeros([go.N * go.N + 1], dtype=np.float32)
        self.child_W = np.zeros([go.N * go.N + 1], dtype=np.float32)
        # save a copy of the original prior before it gets mutated by d-noise.
        self.original_prior = np.zeros([go.N * go.N + 1], dtype=np.float32)
        self.child_prior = np.zeros([go.N * go.N + 1], dtype=np.float32)
        self.children = {}  # map of flattened moves to resulting MCTSNode

    def __repr__(self):
        return "<MCTSNode move=%s, N=%s, to_play=%s>" % (
            self.position.recent[-1:], self.N, self.position.to_play)

    @property
    def child_action_score(self):
        return (self.child_Q * self.position.to_play +
                self.child_U - 1000 * self.illegal_moves)

    @property
    def child_Q(self):
        return self.child_W / (1 + self.child_N)

    @property
    def child_U(self):
        return ((2.0 * (math.log(
                (1.0 + self.N + FLAGS.c_puct_base) / FLAGS.c_puct_base)
                       + FLAGS.c_puct_init)) * math.sqrt(max(1, self.N - 1)) *
                self.child_prior / (1 + self.child_N))

    @property
    def Q(self):
        return self.W / (1 + self.N)

    @property
    def N(self):
        return self.parent.child_N[self.fmove]

    @N.setter
    def N(self, value):
        self.parent.child_N[self.fmove] = value

    @property
    def W(self):
        return self.parent.child_W[self.fmove]

    @W.setter
    def W(self, value):
        self.parent.child_W[self.fmove] = value
```

binary vector

(P) form paper

prior / 1 + num_visits

```
    @property
    def Q_perspective(self):
        """Return value of position, from perspective of player to play."""
        return self.Q * self.position.to_play

    def select_leaf(self):
        current = self
        pass_move = go.N * go.N
        while True:
            # if a node has never been evaluated, we have no basis to select a child.
            if not current.is_expanded:
                break
            # HACK: if last move was a pass, always investigate double-pass first
            # to avoid situations where we auto-lose by passing too early.
            if (current.position.recent and
                current.position.recent[-1].move is None and
                    current.child_N[pass_move] == 0):
                current = current.maybe_add_child(pass_move)
                continue

            best_move = np.argmax(current.child_action_score)
            current = current.maybe_add_child(best_move)
        return current

    def maybe_add_child(self, fcoord):
        """Adds child node for fcoord if it doesn't already exist, and returns it."""
        if fcoord not in self.children:
            new_position = self.position.play_move(
                coords.from_flat(fcoord))
            self.children[fcoord] = MCTSNode(
                new_position, fmove=fcoord, parent=self)
        return self.children[fcoord]

    def add_virtual_loss(self, up_to):
        """Propagate a virtual loss up to the root node.

        Args:
            up_to: The node to propagate until. (Keep track of this! You'll
                need it to reverse the virtual loss later.)
        """
        self.losses_applied += 1
        # This is a "win" for the current node; hence a loss for its parent node
        # who will be deciding whether to investigate this node again.
        loss = self.position.to_play
        self.W += loss
        if self.parent is None or self is up_to:
            return
        self.parent.add_virtual_loss(up_to)

    def revert_virtual_loss(self, up_to):
        self.losses_applied -= 1
        revert = -1 * self.position.to_play
        self.W += revert
        if self.parent is None or self is up_to:
            return
        self.parent.revert_virtual_loss(up_to)

    def incorporate_results(self, move_probabilities, value, up_to):
        assert move_probabilities.shape == (go.N * go.N + 1,)
        # A finished game should not be going through this code path - should
        # directly call backup_value() on the result of the game.
        assert not self.position.is_game_over()

        # If a node was picked multiple times (despite vlosses), we shouldn't
        # expand it more than once.
```

which move has highest visit count

```
        if self.is_expanded:
            return
        self.is_expanded = True

        # Zero out illegal moves.
        move_probs = move_probabilities * (1 - self.illegal_moves)
        scale = sum(move_probs)
        if scale > 0:
            # Re-normalize move_probabilities.
            move_probs *= 1 / scale

        self.original_prior = self.child_prior = move_probs
        # initialize child Q as current node's value, to prevent dynamics where
        # if B is winning, then B will only ever explore 1 move, because the Q
        # estimation will be so much larger than the 0 of the other moves.
        #
        # Conversely, if W is winning, then B will explore all 362 moves before
        # continuing to explore the most favorable move. This is a waste of search.
        #
        # The value seeded here acts as a prior, and gets averaged into Q calculations.
        self.child_W = np.ones([go.N * go.N + 1], dtype=np.float32) * value
        self.backup_value(value, up_to=up_to)

    def backup_value(self, value, up_to):
        """Propagates a value estimation up to the root node.

        Args:
            value: the value to be propagated (1 = black wins, -1 = white wins)
            up_to: the node to propagate until.
        """
        self.N += 1
        self.W += value
        if self.parent is None or self is up_to:
            return
        self.parent.backup_value(value, up_to)

    def is_done(self):
        """True if the last two moves were Pass or if the position is at a move
        greater than the max depth."""
        return self.position.is_game_over() or self.position.n >= FLAGS.max_game_length

    def inject_noise(self):
        epsilon = 1e-5
        legal_moves = (1 - self.illegal_moves) + epsilon
        a = legal_moves * ([FLAGS.dirichlet_noise_alpha] * (go.N * go.N + 1))
        dirichlet = np.random.dirichlet(a)
        self.child_prior = (self.child_prior * (1 - FLAGS.dirichlet_noise_weight) +
                            dirichlet * FLAGS.dirichlet_noise_weight)

    def children_as_pi(self, squash=False):
        """Returns the child visit counts as a probability distribution, pi
        If squash is true, exponentiate the probabilities by a temperature
        slightly larger than unity to encourage diversity in early play and
        hopefully to move away from 3-3s
        """
        probs = self.child_N
        if squash:
            probs = probs ** .98
        sum_probs = np.sum(probs)
        if sum_probs == 0:
            return probs
        return probs / np.sum(probs)

    def best_child(self):
        # Sort by child_N tie break with action score.
        return np.argmax(self.child_N + self.child_action_score / 10000)
```

Masking

```
    def most_visited_path_nodes(self):
        node = self
        output = []
        while node.children:
            node = node.children.get(node.best_child())
            assert node is not None
            output.append(node)
        return output

    def most_visited_path(self):
        output = []
        node = self
        for node in self.most_visited_path_nodes():
            output.append("%s (%d) ==> " % (
                coords.to_gtp(coords.from_flat(node.fmove)), node.N))

        output.append("Q: {:.5f}\n".format(node.Q))
        return ''.join(output)

    def mvp_gg(self):
        """Returns most visited path in go-gui VAR format e.g. 'b r3 w c17..."""
        output = []
        for node in self.most_visited_path_nodes():
            if max(node.child_N) <= 1:
                break
            output.append(coords.to_gtp(coords.from_flat(node.fmove)))
        return ' '.join(output)

    def rank_children(self):
        ranked_children = list(range(go.N * go.N + 1))
        ranked_children.sort(key=lambda i: (
            self.child_N[i], self.child_action_score[i]), reverse=True)
        return ranked_children

    def describe(self):
        ranked_children = self.rank_children()
        soft_n = self.child_N / max(1, sum(self.child_N))
        prior = self.child_prior
        p_delta = soft_n - prior
        p_rel = np.divide(p_delta, prior, out=np.zeros_like(
            p_delta), where=prior != 0)
        # Dump out some statistics
        output = []
        output.append("{q:.4f}\n".format(q=self.Q))
        output.append(self.most_visited_path())
        output.append(
            "move : action    Q     U     P   P-Dir    N  soft-N  p-delta  p-rel")
        for i in ranked_children[:15]:
            if self.child_N[i] == 0:
                break
            output.append("\n{!s:4} : {: .3f} {: .3f} {:.3f} {:.3f} {:.3f} {:5d} {:.4f}
{: .5f} {: .2f}".format(
                coords.to_gtp(coords.from_flat(i)),
                self.child_action_score[i],
                self.child_Q[i],
                self.child_U[i],
                self.child_prior[i],
                self.original_prior[i],
                int(self.child_N[i]),
                soft_n[i],
                p_delta[i],
                p_rel[i]))
        return ''.join(output)
```