```
# 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.
"""Python implementation of selfplay worker.

This worker is used to set up many parallel selfplay instances."""


import random
import os
import socket
import time

from absl import app, flags
from tensorflow import gfile

import coords
import dual_net
import preprocessing
from strategies import MCTSPlayer
import utils

flags.DEFINE_string('load_file', None, 'Path to model save files.')
flags.DEFINE_string('selfplay_dir', None, 'Where to write game data.')
flags.DEFINE_string('holdout_dir', None, 'Where to write held-out game data.')
flags.DEFINE_string('sgf_dir', None, 'Where to write human-readable SGFs.')
flags.DEFINE_float('holdout_pct', 0.05, 'What percent of games to hold out.')
flags.DEFINE_float('resign_disable_pct', 0.05,
                   'What percent of games to disable resign for.')

# From strategies.py
flags.declare_key_flag('verbose')
flags.declare_key_flag('num_readouts')


FLAGS = flags.FLAGS


def play(network):
    """Plays out a self-play match, returning a MCTSPlayer object containing:
        - the final position
        - the n x 362 tensor of floats representing the mcts search probabilities
        - the n-ary tensor of floats representing the original value-net estimate
          where n is the number of moves in the game
    """
    readouts = FLAGS.num_readouts  # defined in strategies.py
    # Disable resign in 5% of games
    if random.random() < FLAGS.resign_disable_pct:
        resign_threshold = -1.0
    else:
        resign_threshold = None

    player = MCTSPlayer(network, resign_threshold=resign_threshold)

    player.initialize_game()
```

```
    # Must run this once at the start to expand the root node.
    first_node = player.root.select_leaf()
    prob, val = network.run(first_node.position)
    first_node.incorporate_results(prob, val, first_node)

    while True:
        start = time.time()
        player.root.inject_noise()
        current_readouts = player.root.N
        # we want to do "X additional readouts", rather than "up to X readouts".
        while player.root.N < current_readouts + readouts:
            player.tree_search()

        if FLAGS.verbose >= 3:
            print(player.root.position)
            print(player.root.describe())

        if player.should_resign():
            player.set_result(-1 * player.root.position.to_play,
                              was_resign=True)
            break
        move = player.pick_move()
        player.play_move(move)
        if player.root.is_done():
            player.set_result(player.root.position.result(), was_resign=False)
            break

        if (FLAGS.verbose >= 2) or (FLAGS.verbose >= 1 and player.root.position.n % 10 ==
9):
            print("Q: {:.5f}".format(player.root.Q))
            dur = time.time() - start
            print("%d: %d readouts, %.3f s/100. (%.2f sec)" % (
                player.root.position.n, readouts, dur / readouts * 100.0, dur),
flush=True)
        if FLAGS.verbose >= 3:
            print("Played >>",
                  coords.to_gtp(coords.from_flat(player.root.fmove)))

    if FLAGS.verbose >= 2:
        utils.dbg("%s: %.3f" % (player.result_string, player.root.Q))
        utils.dbg(player.root.position, player.root.position.score())

    return player


def run_game(load_file, selfplay_dir=None, holdout_dir=None,
             sgf_dir=None, holdout_pct=0.05):
    """Takes a played game and record results and game data."""
    if sgf_dir is not None:
        minimal_sgf_dir = os.path.join(sgf_dir, 'clean')
        full_sgf_dir = os.path.join(sgf_dir, 'full')
        utils.ensure_dir_exists(minimal_sgf_dir)
        utils.ensure_dir_exists(full_sgf_dir)
    if selfplay_dir is not None:
        utils.ensure_dir_exists(selfplay_dir)
        utils.ensure_dir_exists(holdout_dir)

    with utils.logged_timer("Loading weights from %s ... " % load_file):
        network = dual_net.DualNetwork(load_file)

    with utils.logged_timer("Playing game"):
        player = play(network)

    output_name = '{}-{}'.format(int(time.time()), socket.gethostname())
    game_data = player.extract_data()
    if sgf_dir is not None:
```

Run until break

multiple instances in parallel

player = MCTS player(c)

```
        with gfile.GFile(os.path.join(minimal_sgf_dir, '{}.sgf'.format(output_name)), 'w')
as f:
            f.write(player.to_sgf(use_comments=False))
        with gfile.GFile(os.path.join(full_sgf_dir, '{}.sgf'.format(output_name)), 'w') as
f:
            f.write(player.to_sgf())

    tf_examples = preprocessing.make_dataset_from_selfplay(game_data)

    if selfplay_dir is not None:
        # Hold out 5% of games for validation.
        if random.random() < holdout_pct:
            fname = os.path.join(holdout_dir,
                                 "{}.tfrecord.zz".format(output_name))
        else:
            fname = os.path.join(selfplay_dir,
                                 "{}.tfrecord.zz".format(output_name))

        preprocessing.write_tf_examples(fname, tf_examples)


def main(argv):
    """Entry point for running one selfplay game."""
    del argv  # Unused
    flags.mark_flag_as_required('load_file')

    run_game(
        load_file=FLAGS.load_file,
        selfplay_dir=FLAGS.selfplay_dir,
        holdout_dir=FLAGS.holdout_dir,
        holdout_pct=FLAGS.holdout_pct,
        sgf_dir=FLAGS.sgf_dir)


if __name__ == '__main__':
    app.run(main)
```