In [1]:
```
from __future__ import print_function

import gym
from gym import spaces
from gym.utils import seeding

import numpy as np

import itertools
import logging
from six import StringIO
import sys

import torch.nn as nn
import torch.nn.functional as F


import torch
import copy

import matplotlib.pyplot as plt
import os
import shutil
import time

import os.path as osp
import json
import datetime
import tempfile
from collections import defaultdict
from contextlib import contextmanager


print("import done")
```

```
import done
```

In [2]:
```
def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = itertools.tee(iterable)
    next(b, None)
    return zip(a, b)

class IllegalMove(Exception):
    pass

def stack(flat, layers=16):
  larray = []
  for i in range(1, layers + 1):
    ii = 2 ** i
    layer = np.copy(flat)
    layer[layer != ii] = 0
    layer[layer == ii] = 1
    larray.append(layer)

  newstack = np.stack(larray, axis=-1)
  return newstack

class Game2048Env(gym.Env):   # directions 0, 1, 2, 3 are up, right, down, left
    metadata = {'render.modes': ['human', 'ansi']}
    max_steps = 10000

    def __init__(self):
        self.size = 4
        self.w = self.size
        self.h = self.size
        self.squares = self.size * self.size

        self.score = 0

        self.action_space = spaces.Discrete(4)

        layers = self.squares
        self.observation_space = spaces.Box(0, 1, (self.w, self.h, layers), dtype
        self.set_illegal_move_reward(0.)
        self.set_max_tile(None)

        self.max_illegal = 10
        self.num_illegal = 0

        self.seed()


    def _get_info(self, info=None):
        if not info:
            info = {}
        else:
            assert type(info) == dict, 'info should be of type dict!'

        info['highest'] = self.highest()
        info['score'] = self.score
        info['steps'] = self.steps
        return info
```

```
    def seed(self, seed=None):
        self.np_random, seed = seeding.np_random(seed)
        return [seed]

    def set_illegal_move_reward(self, reward):

        self.illegal_move_reward = reward
        self.reward_range = (self.illegal_move_reward, float(2**self.squares))

    def set_max_tile(self, max_tile):

        assert max_tile is None or isinstance(max_tile, int)
        self.max_tile = max_tile

    # Implement gym interface
    def step(self, action):

        logging.debug("Action {}".format(action))
        self.steps += 1
        score = 0
        done = None
        info = {
            'illegal_move': False,
        }
        try:
            score = float(self.move(action))
            self.score += score
            assert score <= 2**(self.w*self.h)
            self.add_tile()
            done = self.isend()
            reward = float(score)
        except IllegalMove as e:
            logging.debug("Illegal move")
            info['illegal_move'] = True
            if self.steps > self.max_steps:
                done = True
            else:
                done = False
            reward = self.illegal_move_reward
            self.num_illegal += 1
            if self.num_illegal >= self.max_illegal:
                done = True

        info = self._get_info(info)

        return self.Matrix, reward, done, info

    def reset(self):
        self.Matrix = np.zeros((self.h, self.w), np.int)
        self.score = 0
        self.steps = 0
        self.num_illegal = 0

        logging.debug("Adding tiles")
        self.add_tile()
        self.add_tile()
```

```
        return self.Matrix, 0, False, self._get_info()

    def render(self, mode='human'):
        outfile = StringIO() if mode == 'ansi' else sys.stdout
        s = 'Score: {}\n'.format(self.score)
        s += 'Highest: {}\n'.format(self.highest())
        npa = np.array(self.Matrix)
        grid = npa.reshape((self.size, self.size))
        s += "{}\n\n".format(grid)
        outfile.write(s)
        return outfile

    # Implement 2048 game
    def add_tile(self):
        possible_tiles = np.array([2, 4])
        tile_probabilities = np.array([0.9, 0.1])
        val = self.np_random.choice(possible_tiles, 1, p=tile_probabilities)[0]
        empties = self.empties()
        assert empties.shape[0]
        empty_idx = self.np_random.choice(empties.shape[0])
        empty = empties[empty_idx]
        logging.debug("Adding %s at %s", val, (empty[0], empty[1]))
        self.set(empty[0], empty[1], val)

    def get(self, x, y):
        return self.Matrix[x, y]

    def set(self, x, y, val):
        self.Matrix[x, y] = val

    def empties(self):
        return np.argwhere(self.Matrix == 0)

    def highest(self):
        return np.max(self.Matrix)

    def move(self, direction, trial=False):
        if not trial:
            if direction == 0:
                logging.debug("Up")
            elif direction == 1:
                logging.debug("Right")
            elif direction == 2:
                logging.debug("Down")
            elif direction == 3:
                logging.debug("Left")

        changed = False
        move_score = 0
        dir_div_two = int(direction / 2)
        dir_mod_two = int(direction % 2)
        shift_direction = dir_mod_two ^ dir_div_two

        rx = list(range(self.w))
        ry = list(range(self.h))
```

```
        if dir_mod_two == 0:
            for y in range(self.h):
                old = [self.get(x, y) for x in rx]
                (new, ms) = self.shift(old, shift_direction)
                move_score += ms
                if old != new:
                    changed = True
                    if not trial:
                        for x in rx:
                            self.set(x, y, new[x])
        else:
            for x in range(self.w):
                old = [self.get(x, y) for y in ry]
                (new, ms) = self.shift(old, shift_direction)
                move_score += ms
                if old != new:
                    changed = True
                    if not trial:
                        for y in ry:
                            self.set(x, y, new[y])
        if changed != True:
            raise IllegalMove

        return move_score

    def combine(self, shifted_row):
        move_score = 0
        combined_row = [0] * self.size
        skip = False
        output_index = 0
        for p in pairwise(shifted_row):
            if skip:
                skip = False
                continue
            combined_row[output_index] = p[0]
            if p[0] == p[1]:
                combined_row[output_index] += p[1]
                move_score += p[0] + p[1]
                # Skip the next thing in the list.
                skip = True
            output_index += 1
        if shifted_row and not skip:
            combined_row[output_index] = shifted_row[-1]

        return (combined_row, move_score)

    def shift(self, row, direction):
        length = len(row)
        assert length == self.size
        assert direction == 0 or direction == 1

        shifted_row = [i for i in row if i != 0]

        if direction:
            shifted_row.reverse()

        (combined_row, move_score) = self.combine(shifted_row)
```

```
        if direction:
            combined_row.reverse()

        assert len(combined_row) == self.size
        return (combined_row, move_score)

    def isend(self):

        if self.max_tile is not None and self.highest() == self.max_tile:
            return True

        if self.steps >= self.max_steps:
            return True

        for direction in range(4):
            try:
                self.move(direction, trial=True)
                # Not the end if we can do any move
                return False
            except IllegalMove:
                pass
        return True

    def get_board(self):
        return self.Matrix

    def set_board(self, new_board):
        self.Matrix = new_board

    print("done1")
```

```
done1
```

In [3]:
```
class SumTree(object):
    data_pointer = 0

    def __init__(self, capacity):
        self.capacity = capacity
        self.tree = np.zeros(2 * capacity - 1)
        self.data = np.zeros(capacity, dtype=object)

    def add(self, p, data):
        tree_idx = self.data_pointer + self.capacity - 1
        self.data[self.data_pointer] = data
        self.update(tree_idx, p)

        self.data_pointer += 1
        if self.data_pointer >= self.capacity:
            self.data_pointer = 0

    def update(self, tree_idx, p):
        change = p - self.tree[tree_idx]
        self.tree[tree_idx] = p

        while tree_idx != 0:
            tree_idx = (tree_idx - 1) // 2
            self.tree[tree_idx] += change

    def get_leaf(self, v):
        parent_idx = 0
        while True:
            cl_idx = 2 * parent_idx + 1
            cr_idx = cl_idx + 1
            if cl_idx >= len(self.tree):
                leaf_idx = parent_idx
                break
            else:
                if v <= self.tree[cl_idx]:
                    parent_idx = cl_idx
                else:
                    v -= self.tree[cl_idx]
                    parent_idx = cr_idx

        data_idx = leaf_idx - self.capacity + 1
        return leaf_idx, self.tree[leaf_idx], self.data[data_idx]

    @property
    def total_p(self):
        return self.tree[0]


class Buffer_PER(object):
    epsilon = 0.01
    alpha = 0.6
    beta = 0.4
    beta_increment_per_sampling = 0.001
    abs_err_upper = 1.

    def __init__(self, capacity):
```

```
        self.tree = SumTree(capacity)

    def store(self, transition):
        max_p = np.max(self.tree.tree[-self.tree.capacity:])
        if max_p == 0:
            max_p = self.abs_err_upper
        self.tree.add(max_p, transition)

    def sample(self, n):
        b_idx, b_memory, ISWeights = np.empty((n,), dtype=np.int32), np.empty((n,
        pri_seg = self.tree.total_p / n
        self.beta = np.min([1., self.beta + self.beta_increment_per_sampling])

        min_prob = np.min(self.tree.tree[-self.tree.capacity:]) / self.tree.total
        for i in range(n):
            a, b = pri_seg * i, pri_seg * (i + 1)
            v = np.random.uniform(a, b)
            idx, p, data = self.tree.get_leaf(v)
            prob = p / self.tree.total_p
            ISWeights[i, 0] = np.power(prob/min_prob, -self.beta)
            b_idx[i], b_memory[i, :] = idx, data

        return b_idx, b_memory, ISWeights

    def batch_update(self, tree_idx, abs_errors):
        abs_errors += self.epsilon
        clipped_errors = np.minimum(abs_errors, self.abs_err_upper)
        ps = np.power(clipped_errors, self.alpha)
        for ti, p in zip(tree_idx, ps):
            self.tree.update(ti, p)


class Buffer():
    def __init__(self,n_features, buffer_type='', capacity=1e4):
        self.memory_size = capacity
        self.n_features = n_features
        self.type = buffer_type
        self.memory_counter = 0

        if self.type == 'priority':
            self.memory = Buffer_PER(capacity=capacity)
        else:
            self.memory = np.zeros((self.memory_size, n_features*2+2))

    def store(self, transition):
        self.memory_counter += 1

        if self.type == 'priority':
            self.memory.store(transition)
        else:
            index = self.memory_counter % self.memory_size
            self.memory[index, :] = transition


    def sample(self, batch_size):
        info = None
        if self.type == 'priority':
```

```
            tree_idx, batch_memory, ISWeights = self.memory.sample(batch_size)
            info = (tree_idx, ISWeights)
        else:
            sample_index = np.random.choice(self.memory_size, size=batch_size)
            batch_memory = self.memory[sample_index, :]

        return batch_memory, info

    def update(self, tree_idx, td_errors):
        assert self.type == 'priority'
        self.memory.batch_update(tree_idx, td_errors)

    print("done2")
```

done2

In [4]:

```
def log2_shaping(s, divide=16):
    s = np.log2(1 + s) / divide
    return s


def check_path_exist(path, verbose=True):
    if not os.path.exists(path):
        os.mkdir(path)
        if verbose:
            print("make the dir {} finished".format(path))
    else:
        if verbose:
            print("the directory {} already exists".format(path))

def running_average(lis, length=5):
    if len(lis) > 10:
        end = len(lis) // length
        lis = lis[:end * length]
        arr = np.array(lis).reshape(-1, length)
        arr = arr.mean(axis=1)

        return list(arr.reshape(-1))
    else:
        return lis


def plot_save(lis, path, title=None, x_label=None, y_label=None):
    dir = path.split("/")[:-1]
    dir = "/".join(dir) + "/"
    check_path_exist(dir, verbose=False)
    plt.figure()
    if type(lis[0]) == list:
        for li in lis:
            plt.plot(li)
    else:
        plt.plot(lis)

    if title:
        plt.title(title)
    if x_label:
        plt.xlabel(x_label)
    if y_label:
        plt.ylabel(y_label)

    plt.savefig(path)
    plt.close("all")


def del_dir_tree(path):
    if os.path.exists(path):
        try:
            shutil.rmtree(path)
        except:
            print("remove path {} failed!".format(path))
```

```
def del_files(path):
    if os.path.isdir(path):
        files = os.listdir(path)
        for file in files:
            os.remove(os.path.join(path, file))
        print("Remove files in {}".format(path))
    elif os.path.isfile(path):
        os.remove(path)
        print("Remove file {}".format(path))
    else:
        print("{} not a file or a directory".format(path))



class Perfomance_Saver():

    def __init__(self, path='performance_data.txt'):
        self.path = path
        self.clear_file()

    def clear_file(self):
        with open(self.path, 'w') as file:
            file.write('clear since :{}\n\n'.format(time.ctime()))
        print("clear file finished")

    def save(self, performance_list, info):
        with open(self.path, 'a+') as file:
            file.writelines("time: {}\n".format(time.ctime()))
            file.writelines("info: {} \n".format(str(info)))
            performance_str = [str(x) + " " for x in performance_list]
            file.writelines(performance_str)
            file.writelines('\n\n')
        print('write to file finished')


class Model_Saver():

    def __init__(self, num=10):
        self.num_max = num
        self.path_list = []

    def save(self, path):
        if len(self.path_list) >= self.num_max:
            os.remove(self.path_list.pop(0))
            print('del surplus modle files')

        self.path_list.append(path)

print("done3")
```

```
done3
```

In [5]:

```
DEBUG = 10
INFO = 20
WARN = 30
ERROR = 40

DISABLED = 50


class KVWriter(object):
    def writekvs(self, kvs):
        raise NotImplementedError


class SeqWriter(object):
    def writeseq(self, seq):
        raise NotImplementedError


class HumanOutputFormat(KVWriter, SeqWriter):
    def __init__(self, filename_or_file):
        if isinstance(filename_or_file, str):
            self.file = open(filename_or_file, 'wt')
            self.own_file = True
        else:
            assert hasattr(
                filename_or_file,
                'read'), 'expected file or str, got %s' % filename_or_file
            self.file = filename_or_file
            self.own_file = False

    def writekvs(self, kvs):
        key2str = {}
        for (key, val) in sorted(kvs.items()):
            if hasattr(val, '__float__'):
                valstr = '%-8.3g' % val
            else:
                valstr = str(val)
            key2str[self._truncate(key)] = self._truncate(valstr)

        if len(key2str) == 0:
            print('WARNING: tried to write empty key-value dict')
            return
        else:
            keywidth = max(map(len, key2str.keys()))
            valwidth = max(map(len, key2str.values()))

        dashes = '-' * (keywidth + valwidth + 7)
        lines = [dashes]
        for (key, val) in sorted(key2str.items(),
                                 key=lambda kv: kv[0].lower()):
            lines.append('| %s%s | %s%s |' % (
                key,
                ' ' * (keywidth - len(key)),
                val,
                ' ' * (valwidth - len(val)),
```

```
            ))
        lines.append(dashes)
        self.file.write('\n'.join(lines) + '\n')

        self.file.flush()

    def _truncate(self, s):
        maxlen = 30
        return s[:maxlen - 3] + '...' if len(s) > maxlen else s

    def writeseq(self, seq):
        seq = list(seq)
        for (i, elem) in enumerate(seq):
            self.file.write(elem)
            if i < len(seq) - 1:
                self.file.write(' ')
        self.file.write('\n')
        self.file.flush()

    def close(self):
        if self.own_file:
            self.file.close()


class JSONOutputFormat(KVWriter):
    def __init__(self, filename):
        self.file = open(filename, 'wt')

    def writekvs(self, kvs):
        for k, v in sorted(kvs.items()):
            if hasattr(v, 'dtype'):
                kvs[k] = float(v)
        self.file.write(json.dumps(kvs) + '\n')
        self.file.flush()

    def close(self):
        self.file.close()


class CSVOutputFormat(KVWriter):
    def __init__(self, filename):
        self.file = open(filename, 'w+t')
        self.keys = []
        self.sep = ','

    def writekvs(self, kvs):
        extra_keys = list(kvs.keys() - self.keys)
        extra_keys.sort()
        if extra_keys:
            self.keys.extend(extra_keys)
            self.file.seek(0)
            lines = self.file.readlines()
            self.file.seek(0)
            for (i, k) in enumerate(self.keys):
                if i > 0:
                    self.file.write(',')
                self.file.write(k)
```

```
            self.file.write('\n')
            for line in lines[1:]:
                self.file.write(line[:-1])
                self.file.write(self.sep * len(extra_keys))
                self.file.write('\n')
        for (i, k) in enumerate(self.keys):
            if i > 0:
                self.file.write(',')
            v = kvs.get(k)
            if v is not None:
                self.file.write(str(v))
        self.file.write('\n')
        self.file.flush()

    def close(self):
        self.file.close()


class TensorBoardOutputFormat(KVWriter):

    def __init__(self, dir):
        os.makedirs(dir, exist_ok=True)
        del_files(dir)
        self.dir = dir
        self.step = 1
        prefix = 'events'
        path = osp.join(osp.abspath(dir), prefix)
        import tensorflow as tf
        from tensorflow.python import pywrap_tensorflow
        from tensorflow.core.util import event_pb2
        from tensorflow.python.util import compat
        self.tf = tf
        self.event_pb2 = event_pb2
        self.pywrap_tensorflow = pywrap_tensorflow
        self.writer = pywrap_tensorflow.EventsWriter(compat.as_bytes(path))

    def writekvs(self, kvs):
        def summary_val(k, v):
            kwargs = {'tag': k, 'simple_value': float(v)}
            return self.tf.Summary.Value(**kwargs)

        summary = self.tf.Summary(
            value=[summary_val(k, v) for k, v in kvs.items()])
        event = self.event_pb2.Event(wall_time=time.time(), summary=summary)
        event.step = self.step
        self.writer.WriteEvent(event)
        self.writer.Flush()
        self.step += 1

    def close(self):
        if self.writer:
            self.writer.Close()
            self.writer = None


def make_output_format(format, ev_dir, log_suffix=''):
    os.makedirs(ev_dir, exist_ok=True)
```

```
    if format == 'stdout':
        return HumanOutputFormat(sys.stdout)
    elif format == 'log':
        return HumanOutputFormat(osp.join(ev_dir, 'log%s.txt' % log_suffix))
    elif format == 'json':
        return JSONOutputFormat(
            osp.join(ev_dir, 'progress%s.json' % log_suffix))
    elif format == 'csv':
        return CSVOutputFormat(osp.join(ev_dir, 'progress%s.csv' % log_suffix))
    elif format == 'tensorboard':
        return TensorBoardOutputFormat(osp.join(ev_dir, 'tb%s' % log_suffix))
    else:
        raise ValueError('Unknown format specified: %s' % (format, ))



def logkv(key, val):

    get_current().logkv(key, val)


def logkv_mean(key, val):

    get_current().logkv_mean(key, val)


def logkvs(d):

    for (k, v) in d.items():
        logkv(k, v)


def dumpkvs():

    return get_current().dumpkvs()


def getkvs():
    return get_current().name2val


def log(*args, level=INFO):

    get_current().log(*args, level=level)


def debug(*args):
    log(*args, level=DEBUG)


def info(*args):
    log(*args, level=INFO)


def warn(*args):
    log(*args, level=WARN)
```

```
def error(*args):
    log(*args, level=ERROR)


def set_level(level):

    get_current().set_level(level)


def set_comm(comm):
    get_current().set_comm(comm)


def get_dir():

    return get_current().get_dir()


record_tabular = logkv
dump_tabular = dumpkvs


@contextmanager
def profile_kv(scopename):
    logkey = 'wait_' + scopename
    tstart = time.time()
    try:
        yield
    finally:
        get_current().name2val[logkey] += time.time() - tstart


def profile(n):

    def decorator_with_name(func):
        def func_wrapper(*args, **kwargs):
            with profile_kv(n):
                return func(*args, **kwargs)

        return func_wrapper

    return decorator_with_name


def get_current():
    if Logger.CURRENT is None:
        _configure_default_logger()

    return Logger.CURRENT


class Logger(object):
    DEFAULT = None
    CURRENT = None

    def __init__(self, dir, output_formats, comm=None):
```

```
        self.name2val = defaultdict(float)
        self.name2cnt = defaultdict(int)
        self.level = INFO
        self.dir = dir
        self.output_formats = output_formats
        self.comm = comm


    def logkv(self, key, val):
        self.name2val[key] = val

    def logkv_mean(self, key, val):
        oldval, cnt = self.name2val[key], self.name2cnt[key]
        self.name2val[key] = oldval * cnt / (cnt + 1) + val / (cnt + 1)
        self.name2cnt[key] = cnt + 1

    def dumpkvs(self):
        if self.comm is None:
            d = self.name2val
        else:
            from baselines.common import mpi_util
            d = mpi_util.mpi_weighted_mean(
                self.comm, {
                    name: (val, self.name2cnt.get(name, 1))
                    for (name, val) in self.name2val.items()
                })
            if self.comm.rank != 0:
                d['dummy'] = 1
        out = d.copy()
        for fmt in self.output_formats:
            if isinstance(fmt, KVWriter):
                fmt.writekvs(d)
        self.name2val.clear()
        self.name2cnt.clear()
        return out

    def log(self, *args, level=INFO):
        if self.level <= level:
            self._do_log(args)


    def set_level(self, level):
        self.level = level

    def set_comm(self, comm):
        self.comm = comm

    def get_dir(self):
        return self.dir

    def close(self):
        for fmt in self.output_formats:
            fmt.close()


    def _do_log(self, args):
        for fmt in self.output_formats:
```

```
            if isinstance(fmt, SeqWriter):
                fmt.writeseq(map(str, args))


def get_rank_without_mpi_import():

    for varname in ['PMI_RANK', 'OMPI_COMM_WORLD_RANK']:
        if varname in os.environ:
            return int(os.environ[varname])
    return 0


def configure(dir=None, format_strs=None, comm=None, log_suffix=''):

    if dir is None:
        dir = os.getenv('OPENAI_LOGDIR')
    if dir is None:
        dir = osp.join(
            tempfile.gettempdir(),
            datetime.datetime.now().strftime("openai-%Y-%m-%d-%H-%M-%S-%f"))
    assert isinstance(dir, str)
    dir = os.path.expanduser(dir)
    os.makedirs(os.path.expanduser(dir), exist_ok=True)

    rank = get_rank_without_mpi_import()
    if rank > 0:
        log_suffix = log_suffix + "-rank%03i" % rank

    if format_strs:
        format_strs = format_strs.split(",")
    else:
        raise TypeError("format error")

    output_formats = [
        make_output_format(f, dir, log_suffix) for f in format_strs
    ]

    Logger.CURRENT = Logger(dir=dir, output_formats=output_formats, comm=comm)
    if output_formats:
        log('Logging to %s' % dir)


def _configure_default_logger():
    configure()
    Logger.DEFAULT = Logger.CURRENT


def reset():
    if Logger.CURRENT is not Logger.DEFAULT:
        Logger.CURRENT.close()
        Logger.CURRENT = Logger.DEFAULT
        log('Reset logger')


@contextmanager
def scoped_configure(dir=None, format_strs=None, comm=None):
    prevlogger = Logger.CURRENT
```

```
    configure(dir=dir, format_strs=format_strs, comm=comm)
    try:
        yield
    finally:
        Logger.CURRENT.close()
        Logger.CURRENT = prevlogger



def _demo():
    info("hello")
    debug("shouldn't appear")
    set_level(DEBUG)
    debug("should appear")
    dir = "/tmp/testlogging"
    if os.path.exists(dir):
        shutil.rmtree(dir)
    configure(dir=dir)
    logkv("a", 3)
    logkv("b", 2.5)
    dumpkvs()
    logkv("b", -2.5)
    logkv("a", 5.5)
    dumpkvs()
    info("^^^ should see a = 5.5")
    logkv_mean("b", -22.5)
    logkv_mean("b", -44.4)
    logkv("a", 5.5)
    dumpkvs()
    info("^^^ should see b = -33.3")

    logkv("b", -2.5)
    dumpkvs()

    logkv("a", "value")
    dumpkvs()

def read_json(fname):
    import pandas
    ds = []
    with open(fname, 'rt') as fh:
        for line in fh:
            ds.append(json.loads(line))
    return pandas.DataFrame(ds)


def read_csv(fname):
    import pandas
    return pandas.read_csv(fname, index_col=None, comment='#')


def read_tb(path):
    import pandas
    import numpy as np
    from glob import glob
    import tensorflow as tf
    if osp.isdir(path):
```

```
    fnames = glob(osp.join(path, "events.*"))
elif osp.basename(path).startswith("events."):
    fnames = [path]
else:
    raise NotImplementedError(
        "Expected tensorboard file or directory containing them. Got %s" %
        path)
tag2pairs = defaultdict(list)
maxstep = 0
for fname in fnames:
    for summary in tf.train.summary_iterator(fname):
        if summary.step > 0:
            for v in summary.summary.value:
                pair = (summary.step, v.simple_value)
                tag2pairs[v.tag].append(pair)
            maxstep = max(summary.step, maxstep)
data = np.empty((maxstep, len(tag2pairs)))
data[:] = np.nan
tags = sorted(tag2pairs.keys())
for (colidx, tag) in enumerate(tags):
    pairs = tag2pairs[tag]
    for (step, value) in pairs:
        data[step - 1, colidx] = value
return pandas.DataFrame(data, columns=tags)
```

In [6]:

```
class CNN_Net(nn.Module):
    def __init__(self, input_len, output_num, conv_size=(32, 64), fc_size=(1024,
        super(CNN_Net, self).__init__()
        self.input_len = input_len
        self.output_num = output_num
        self.out_softmax = out_softmax

        self.conv1 = nn.Sequential(
            nn.Conv2d(1, conv_size[0], kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(conv_size[0], conv_size[1], kernel_size=3, stride=1, paddin
            nn.ReLU(inplace=True),
        )

        self.fc1 = nn.Linear(conv_size[1] * self.input_len * self.input_len, fc_s
        self.fc2 = nn.Linear(fc_size[0], fc_size[1])
        self.head = nn.Linear(fc_size[1], self.output_num)

    def forward(self, x):
        x = x.reshape(-1,1,self.input_len, self.input_len)
        x = self.conv1(x)
        x = self.conv2(x)
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))

        output = self.head(x)
        if self.out_softmax:
            output = F.softmax(output, dim=1)
        return output



class FC_Net(nn.Module):
    def __init__(self, input_num, output_num, fc_size=(1024, 128), out_softmax=Fa
        super(FC_Net, self).__init__()
        self.input_num = input_num
        self.output_num = output_num
        self.out_softmax = out_softmax

        self.fc1 = nn.Linear(self.input_num, fc_size[0])
        self.fc2 = nn.Linear(fc_size[0], fc_size[1])
        self.head = nn.Linear(fc_size[1], self.output_num)

    def forward(self, x):
        x = x.reshape(-1, self.input_num)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))

        output = self.head(x)
        if self.out_softmax:
            output = F.softmax(output, dim=1)
        return output
```

In [7]:
```
import random


class BaseAgent():
    def act(self, state):
        raise NotImplementedError


class RandomAgent(BaseAgent):
    def act(self, state):
        return random.randint(0, 3)


if __name__ == "__main__":
    import time
    import numpy as np

    def run(ifrender=False):
        agent = RandomAgent()
        env = Game2048Env()
        state, reward, done, info = env.reset()
        if ifrender:
            env.render()

        start = time.time()
        while True:
            action = agent.act(state)
            # print('action: {}'.format(action))
            state, reward, done, info = env.step(action)
            if ifrender:
                env.render()
            if done:
                print('\nfinished, info:{}'.format(info))
                break

        end = time.time()
        print('episode time:{} s\n'.format(end - start))
        return end - start, info['highest'], info['score'], info['steps']

    time_lis, highest_lis, score_lis, steps_lis = [], [], [], []
    for i in range(1000):
        t, highest, score, steps = run()
        time_lis.append(t)
        highest_lis.append(highest)
        score_lis.append(score)
        steps_lis.append(steps)

    print('eval result:\naverage episode time:{} s, average highest score:{}, ave
```

```
s': 74}
```

```
episode time:0.013960599899291992 s


finished, info:{'illegal_move': True, 'highest': 32, 'score': 336.0, 'step
s': 62}
episode time:0.011934518814086914 s


finished, info:{'illegal_move': True, 'highest': 128, 'score': 944.0, 'step
s': 103}
episode time:0.02393651008605957 s


finished, info:{'illegal_move': True, 'highest': 64, 'score': 704.0, 'step
s': 91}
episode time:0.01994609832763672 s

eval result:
```

In [8]:

```
class DQN():
    batch_size = 128
    lr = 1e-4
    epsilon = 0.15
    memory_capacity =  int(1e4)
    gamma = 0.99
    q_network_iteration = 200
    save_path = "./save/"
    soft_update_theta = 0.1
    clip_norm_max = 1
    train_interval = 5
    conv_size = (32, 64)   # num filters
    fc_size = (512, 128)

    def __init__(self, num_state, num_action, enable_double=False, enable_priorit
        super(DQN, self).__init__()
        self.num_state = num_state
        self.num_action = num_action
        self.state_len = int(np.sqrt(self.num_state))
        self.enable_double = enable_double
        self.enable_priority = enable_priority

        self.eval_net, self.target_net = CNN_Net(self.state_len, num_action,self.


        self.learn_step_counter = 0
        self.buffer = Buffer(self.num_state, 'priority', self.memory_capacity)

        self.initial_epsilon = self.epsilon
        self.optimizer = torch.optim.Adam(self.eval_net.parameters(), lr=self.lr)


    def select_action(self, state, random=False, deterministic=False):
        state = torch.unsqueeze(torch.FloatTensor(state), 0)
        if not random and np.random.random() > self.epsilon or deterministic:
            action_value = self.eval_net.forward(state)
            action = torch.max(action_value.reshape(-1,4), 1)[1].data.numpy()
        else:
            action = np.random.randint(0,self.num_action)
        return action


    def store_transition(self, state, action, reward, next_state):
        state = state.reshape(-1)
        next_state = next_state.reshape(-1)

        transition = np.hstack((state, [action, reward], next_state))
        self.buffer.store(transition)


    def update(self):

        if self.learn_step_counter % self.q_network_iteration ==0 and self.learn_
            for p_e, p_t in zip(self.eval_net.parameters(), self.target_net.param
```

```
                p_t.data = self.soft_update_theta * p_e.data + (1 - self.soft_upd

        self.learn_step_counter+=1


        if self.enable_priority:
            batch_memory, (tree_idx, ISWeights) = self.buffer.sample(self.batch_s
        else:
            batch_memory, _ = self.buffer.sample(self.batch_size)

        batch_state = torch.FloatTensor(batch_memory[:, :self.num_state])
        batch_action = torch.LongTensor(batch_memory[:, self.num_state: self.num_
        batch_reward = torch.FloatTensor(batch_memory[:, self.num_state+1: self.n
        batch_next_state = torch.FloatTensor(batch_memory[:,-self.num_state:])


        q_eval_total = self.eval_net(batch_state)
        q_eval = q_eval_total.gather(1, batch_action)
        q_next = self.target_net(batch_next_state).detach()

        if self.enable_double:
            q_eval_argmax = q_eval_total.max(1)[1].view(self.batch_size, 1)
            q_max = q_next.gather(1, q_eval_argmax).view(self.batch_size, 1)
        else:
            q_max = q_next.max(1)[0].view(self.batch_size, 1)

        q_target = batch_reward + self.gamma * q_max

        if self.enable_priority:
            abs_errors = (q_target - q_eval.data).abs()
            self.buffer.update(tree_idx, abs_errors)

            loss = (q_target - q_eval).pow(2).mean()


        else:
            loss = F.mse_loss(q_eval, q_target)


        self.optimizer.zero_grad()
        loss.backward()
        nn.utils.clip_grad_norm_(self.eval_net.parameters(), self.clip_norm_max)
        self.optimizer.step()

        return loss


    def save(self, path=None, name='dqn_net.pkl'):
        path = self.save_path if not path else path
        check_path_exist(path)
        torch.save(self.eval_net.state_dict(), path + name)

    def load(self, path=None, name='dqn_net.pkl'):
        path = self.save_path if not path else path
        self.eval_net.load_state_dict(torch.load(path + name))
```

```
    def epsilon_decay(self, episode, total_episode):
        self.epsilon = self.initial_epsilon * (1 - episode / total_episode)
```

In [9]:

```
train_episodes = 2000
test_episodes = 20
ifrender = False
eval_interval = 25
epsilon_decay_interval = 100
log_interval = 100



def train():
    episodes = train_episodes
#     logger = Logger.configure(dir="./log/", format_strs="stdout,tensorboard,log
    agent = DQN(num_state=16, num_action=4)
    env = Game2048Env()
    print("hey1")

    pf_saver = Perfomance_Saver()
    print("hey2")
    model_saver = Model_Saver(num=10)
    print("hey3")

    eval_max_score = 0
    for i in range(episodes):
        print(i)
        state, reward, done, info = env.reset()
        state = log2_shaping(state)

        start = time.time()
        loss = None
        while True:
            if agent.buffer.memory_counter <= agent.memory_capacity:
                action = agent.select_action(state, random=True)
            else:
                action = agent.select_action(state)

            next_state, reward, done, info = env.step(action)
            next_state = log2_shaping(next_state)
            reward = log2_shaping(reward, divide=1)

            agent.store_transition(state, action, reward, next_state)
            state = next_state

            if ifrender:
                env.render()

            if agent.buffer.memory_counter % agent.train_interval == 0 and agent.
                loss = agent.update()

            if done:
#                 if i % log_interval == 0:
#                     if loss:
#                         print('loss', loss)
#                     print('training progress', (i+1) / episodes)
#                     print('episode reward', info['score'])
```

```
#                     print('episode steps', info['steps'])
#                     print('highest', info['highest'])
#                     print('epsilon', agent.epsilon)

#                     loss = None

                if i % epsilon_decay_interval == 0:   # episilon decay
                    agent.epsilon_decay(i, episodes)
                break

        end = time.time()
#         print('episode time:{} s\n'.format(end - start))

        # eval
        if i % eval_interval == 0 and i:
            eval_info = test(episodes=test_episodes, agent=agent)
            average_score, max_score, score_lis = eval_info['mean'], eval_info['m

            pf_saver.save(score_lis, info="episode:{}".format(i))

            if int(average_score) > eval_max_score:
                eval_max_score = int(average_score)
                name = 'dqn_{}.pkl'.format(int(eval_max_score))
                agent.save(name=name)
                model_saver.save("./save/" + name)

            print('eval average score', average_score)
            print('eval max socre', max_score)



def test(episodes=20, agent=None, load_path=None, ifrender=False, log=False):
#     if log:
#         logger.configure(dir="./log/", format_strs="stdout")
    if agent is None:
        agent = DQN(num_state=16, num_action=4)
        if load_path:
            agent.load(load_path)
        else:
            agent.load()

    env = Game2048Env()
    score_list = []
    highest_list = []

    for i in range(episodes):
        state, _, done, info = env.reset()
        state = log2_shaping(state)

        start = time.time()
        while True:
            action = agent.select_action(state, deterministic=True)
            next_state, _, done, info = env.step(action)
            next_state = log2_shaping(next_state)
            state = next_state

            if ifrender:
```

```
                env.render()

            if done:
#                 if log:
#                     print('episode number', i + 1)
#                     print('episode reward', info['score'])
#                     print('episode steps', info['steps'])
#                     print('highest', info['highest'])
                break

        end = time.time()
#         if log:
#             print('episode time:{} s\n'.format(end - start))

        score_list.append(info['score'])
        highest_list.append(info['highest'])

    print('mean score:{}, mean highest:{}'.format(np.mean(score_list), np.mean(hi
    print('max score:{}, max hightest:{}'.format(np.max(score_list), np.max(highe
    result_info = {'mean':np.mean(score_list), 'max':np.max(score_list), 'list':s
    return result_info


if __name__ == "__main__":
     #test(episodes=test_episodes, ifrender=ifrender)
    train()
```

```
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
```

In [ ]:

In [ ]:

In [ ]:

In [ ]: