/\*\*

\* Temporal Difference Learning Demo for Game 2048

\* use 'g++ -std=c++0x -O3 -g -o 2048 2048.cpp' to compile the source

\* https://github.com/moporgic/TDL2048-Demo

\*

\* Computer Games and Intelligence (CGI) Lab, NCTU, Taiwan

\* http://www.aigames.nctu.edu.tw

\*

\* References:

\* [1] Szubert, Marcin, and Wojciech Jaśkowski. "Temporal difference learning of

\* n-tuple networks for the game 2048." Computational Intelligence and Games

\* (CIG), 2014 IEEE Conference on. IEEE, 2014. [2] Wu, I-Chen, et al.

\* "Multi-stage temporal difference learning for 2048." Technologies and

\* Applications of Artificial Intelligence. Springer International Publishing,

\* 2014. 366-378. [3] Oka, Kazuto, and Kiminori Matsuzaki. "Systematic selection

\* of n-tuple networks for 2048." International Conference on Computers and

\* Games. Springer International Publishing, 2016.

\*/

#include <algorithm>

#include <array>

#include <cmath>

#include <fstream>

#include <functional>

#include <iostream>

#include <iterator>

#include <limits>

#include <numeric>

#include <sstream>

#include <string>

#include <vector>

/\*\*

\* output streams

\* to enable debugging (more output), just change the line to 'std::ostream&

\* debug = std::cout;'

\*/

std::ostream &info = std::cout;

std::ostream &error = std::cerr;

std::ostream &debug = \*(new std::ofstream);

/\*\*

\* 64-bit bitboard implementation for 2048

\*

\* index:

\* 0 1 2 3

\* 4 5 6 7

\* 8 9 10 11

\* 12 13 14 15

\*

\* note that the 64-bit value is little endian

\* therefore a board with raw value 0x4312752186532731ull would be

\* +------------------------+

\* | 2 8 128 4|

\* | 8 32 64 256|

\* | 2 4 32 128|

\* | 4 2 8 16|

\* +------------------------+

\*

\*/

class board {

public:

board(uint64\_t raw = 0) : raw(raw) {}

board(const board &b) = default;

board &operator=(const board &b) = default;

operator uint64\_t() const { return raw; }

/\*\*

\* get a 16-bit row

\*/

int fetch(int i) const { return ((raw >> (i << 4)) & 0xffff); }

/\*\*

\* set a 16-bit row

\*/

void place(int i, int r) {

raw = (raw & ~(0xffffULL << (i << 4))) | (uint64\_t(r & 0xffff) << (i << 4));

}

/\*\*

\* get a 4-bit tile

\*/

int at(int i) const { return (raw >> (i << 2)) & 0x0f; }

/\*\*

\* set a 4-bit tile

\*/

void set(int i, int t) {

raw = (raw & ~(0x0fULL << (i << 2))) | (uint64\_t(t & 0x0f) << (i << 2));

}

public:

bool operator==(const board &b) const { return raw == b.raw; }

bool operator<(const board &b) const { return raw < b.raw; }

bool operator!=(const board &b) const { return !(\*this == b); }

bool operator>(const board &b) const { return b < \*this; }

bool operator<=(const board &b) const { return !(b < \*this); }

bool operator>=(const board &b) const { return !(\*this < b); }

private:

/\*\*

\* the lookup table for moving board

\*/

struct lookup {

int raw; // base row (16-bit raw)

int left; // left operation

int right; // right operation

int score; // merge reward

void init(int r) {

raw = r;

int V[4] = {(r >> 0) & 0x0f, (r >> 4) & 0x0f, (r >> 8) & 0x0f,

(r >> 12) & 0x0f};

int L[4] = {V[0], V[1], V[2], V[3]};

int R[4] = {V[3], V[2], V[1], V[0]}; // mirrored

score = mvleft(L);

left = ((L[0] << 0) | (L[1] << 4) | (L[2] << 8) | (L[3] << 12));

score = mvleft(R);

std::reverse(R, R + 4);

right = ((R[0] << 0) | (R[1] << 4) | (R[2] << 8) | (R[3] << 12));

}

void move\_left(uint64\_t &raw, int &sc, int i) const {

raw |= uint64\_t(left) << (i << 4);

sc += score;

}

void move\_right(uint64\_t &raw, int &sc, int i) const {

raw |= uint64\_t(right) << (i << 4);

sc += score;

}

static int mvleft(int row[]) {

int top = 0;

int tmp = 0;

int score = 0;

for (int i = 0; i < 4; i++) {

int tile = row[i];

if (tile == 0)

continue;

row[i] = 0;

if (tmp != 0) {

if (tile == tmp) {

tile = tile + 1;

row[top++] = tile;

score += (1 << tile);

tmp = 0;

} else {

row[top++] = tmp;

tmp = tile;

}

} else {

tmp = tile;

}

}

if (tmp != 0)

row[top] = tmp;

return score;

}

lookup() {

static int row = 0;

init(row++);

}

static const lookup &find(int row) {

static const lookup cache[65536];

return cache[row];

}

};

public:

/\*\*

\* reset to initial state (2 random tile on board)

\*/

void init() {

raw = 0;

popup();

popup();

}

/\*\*

\* add a new random tile on board, or do nothing if the board is full

\* 2-tile: 90%

\* 4-tile: 10%

\*/

void popup() {

int space[16], num = 0;

for (int i = 0; i < 16; i++)

if (at(i) == 0) {

space[num++] = i;

}

if (num)

set(space[rand() % num], rand() % 10 ? 1 : 2);

}

/\*\*

\* apply an action to the board

\* return the reward gained by the action, or -1 if the action is illegal

\*/

int move(int opcode) {

switch (opcode) {

case 0:

return move\_up();

case 1:

return move\_right();

case 2:

return move\_down();

case 3:

return move\_left();

default:

return -1;

}

}

int move\_left() {

uint64\_t move = 0;

uint64\_t prev = raw;

int score = 0;

lookup::find(fetch(0)).move\_left(move, score, 0);

lookup::find(fetch(1)).move\_left(move, score, 1);

lookup::find(fetch(2)).move\_left(move, score, 2);

lookup::find(fetch(3)).move\_left(move, score, 3);

raw = move;

return (move != prev) ? score : -1;

}

int move\_right() {

uint64\_t move = 0;

uint64\_t prev = raw;

int score = 0;

lookup::find(fetch(0)).move\_right(move, score, 0);

lookup::find(fetch(1)).move\_right(move, score, 1);

lookup::find(fetch(2)).move\_right(move, score, 2);

lookup::find(fetch(3)).move\_right(move, score, 3);

raw = move;

return (move != prev) ? score : -1;

}

int move\_up() {

rotate\_right();

int score = move\_right();

rotate\_left();

return score;

}

int move\_down() {

rotate\_right();

int score = move\_left();

rotate\_left();

return score;

}

/\*\*

\* swap row and column

\* +------------------------+ +------------------------+

\* | 2 8 128 4| | 2 8 2 4|

\* | 8 32 64 256| | 8 32 4 2|

\* | 2 4 32 128| ----> | 128 64 32 8|

\* | 4 2 8 16| | 4 256 128 16|

\* +------------------------+ +------------------------+

\*/

void transpose() {

raw = (raw & 0xf0f00f0ff0f00f0fULL) |

((raw & 0x0000f0f00000f0f0ULL) << 12) |

((raw & 0x0f0f00000f0f0000ULL) >> 12);

raw = (raw & 0xff00ff0000ff00ffULL) |

((raw & 0x00000000ff00ff00ULL) << 24) |

((raw & 0x00ff00ff00000000ULL) >> 24);

}

/\*\*

\* horizontal reflection

\* +------------------------+ +------------------------+

\* | 2 8 128 4| | 4 128 8 2|

\* | 8 32 64 256| | 256 64 32 8|

\* | 2 4 32 128| ----> | 128 32 4 2|

\* | 4 2 8 16| | 16 8 2 4|

\* +------------------------+ +------------------------+

\*/

void mirror() {

raw = ((raw & 0x000f000f000f000fULL) << 12) |

((raw & 0x00f000f000f000f0ULL) << 4) |

((raw & 0x0f000f000f000f00ULL) >> 4) |

((raw & 0xf000f000f000f000ULL) >> 12);

}

/\*\*

\* vertical reflection

\* +------------------------+ +------------------------+

\* | 2 8 128 4| | 4 2 8 16|

\* | 8 32 64 256| | 2 4 32 128|

\* | 2 4 32 128| ----> | 8 32 64 256|

\* | 4 2 8 16| | 2 8 128 4|

\* +------------------------+ +------------------------+

\*/

void flip() {

raw = ((raw & 0x000000000000ffffULL) << 48) |

((raw & 0x00000000ffff0000ULL) << 16) |

((raw & 0x0000ffff00000000ULL) >> 16) |

((raw & 0xffff000000000000ULL) >> 48);

}

/\*\*

\* rotate the board clockwise by given times

\*/

void rotate(int r = 1) {

switch (((r % 4) + 4) % 4) {

default:

case 0:

break;

case 1:

rotate\_right();

break;

case 2:

reverse();

break;

case 3:

rotate\_left();

break;

}

}

void rotate\_right() {

transpose();

mirror();

} // clockwise

void rotate\_left() {

transpose();

flip();

} // counterclockwise

void reverse() {

mirror();

flip();

}

public:

friend std::ostream &operator<<(std::ostream &out, const board &b) {

char buff[32];

out << "+------------------------+" << std::endl;

for (int i = 0; i < 16; i += 4) {

snprintf(buff, sizeof(buff), "|%6u%6u%6u%6u|",

(1 << b.at(i + 0)) & -2u, // use -2u (0xff...fe) to remove the

// unnecessary 1 for (1 << 0)

(1 << b.at(i + 1)) & -2u, (1 << b.at(i + 2)) & -2u,

(1 << b.at(i + 3)) & -2u);

out << buff << std::endl;

}

out << "+------------------------+" << std::endl;

return out;

}

private:

uint64\_t raw;

};

/\*\*

\* feature and weight table for temporal difference learning

\*/

class feature {

public:

feature(size\_t len) : length(len), weight(alloc(len)) {}

feature(feature &&f) : length(f.length), weight(f.weight) {

f.weight = nullptr;

}

feature(const feature &f) = delete;

feature &operator=(const feature &f) = delete;

virtual ~feature() { delete[] weight; }

float &operator[](size\_t i) { return weight[i]; }

float operator[](size\_t i) const { return weight[i]; }

size\_t size() const { return length; }

public: // should be implemented

/\*\*

\* estimate the value of a given board

\*/

virtual float estimate(const board &b) const = 0;

/\*\*

\* update the value of a given board, and return its updated value

\*/

virtual float update(const board &b, float u) = 0;

/\*\*

\* get the name of this feature

\*/

virtual std::string name() const = 0;

public:

/\*\*

\* dump the detail of weight table of a given board

\*/

virtual void dump(const board &b, std::ostream &out = info) const {

out << b << "estimate = " << estimate(b) << std::endl;

}

friend std::ostream &operator<<(std::ostream &out, const feature &w) {

std::string name = w.name();

int len = name.length();

out.write(reinterpret\_cast<char \*>(&len), sizeof(int));

out.write(name.c\_str(), len);

float \*weight = w.weight;

size\_t size = w.size();

out.write(reinterpret\_cast<char \*>(&size), sizeof(size\_t));

out.write(reinterpret\_cast<char \*>(weight), sizeof(float) \* size);

return out;

}

friend std::istream &operator>>(std::istream &in, feature &w) {

std::string name;

int len = 0;

in.read(reinterpret\_cast<char \*>(&len), sizeof(int));

name.resize(len);

in.read(&name[0], len);

if (name != w.name()) {

error << "unexpected feature: " << name << " (" << w.name()

<< " is expected)" << std::endl;

std::exit(1);

}

float \*weight = w.weight;

size\_t size;

in.read(reinterpret\_cast<char \*>(&size), sizeof(size\_t));

if (size != w.size()) {

error << "unexpected feature size " << size << "for " << w.name();

error << " (" << w.size() << " is expected)" << std::endl;

std::exit(1);

}

in.read(reinterpret\_cast<char \*>(weight), sizeof(float) \* size);

if (!in) {

error << "unexpected end of binary" << std::endl;

std::exit(1);

}

return in;

}

protected:

static float \*alloc(size\_t num) {

static size\_t total = 0;

static size\_t limit = (1 << 30) / sizeof(float); // 1G memory

try {

total += num;

if (total > limit)

throw std::bad\_alloc();

return new float[num]();

} catch (std::bad\_alloc &) {

error << "memory limit exceeded" << std::endl;

std::exit(-1);

}

return nullptr;

}

size\_t length;

float \*weight;

};

/\*\*

\* the pattern feature

\* including isomorphic (rotate/mirror)

\*

\* index:

\* 0 1 2 3

\* 4 5 6 7

\* 8 9 10 11

\* 12 13 14 15

\*

\* usage:

\* pattern({ 0, 1, 2, 3 })

\* pattern({ 0, 1, 2, 3, 4, 5 })

\*/

class pattern : public feature {

public:

pattern(const std::vector<int> &p, int iso = 8)

: feature(1 << (p.size() \* 4)), iso\_last(iso) {

if (p.empty()) {

error << "no pattern defined" << std::endl;

std::exit(1);

}

/\*\*

\* isomorphic patterns can be calculated by board

\*

\* take pattern { 0, 1, 2, 3 } as an example

\* apply the pattern to the original board (left), we will get 0x1372

\* if we apply the pattern to the clockwise rotated board (right), we will

\* get 0x2131, which is the same as applying pattern { 12, 8, 4, 0 } to the

\* original board { 0, 1, 2, 3 } and { 12, 8, 4, 0 } are isomorphic patterns

\* +------------------------+ +------------------------+

\* | 2 8 128 4| | 4 2 8 2|

\* | 8 32 64 256| | 2 4 32 8|

\* | 2 4 32 128| ----> | 8 32 64 128|

\* | 4 2 8 16| | 16 128 256 4|

\* +------------------------+ +------------------------+

\*

\* therefore if we make a board whose value is 0xfedcba9876543210ull (the

\* same as index) we would be able to use the above method to calculate its

\* 8 isomorphisms

\*/

for (int i = 0; i < 8; i++) {

board idx = 0xfedcba9876543210ull;

if (i >= 4)

idx.mirror();

idx.rotate(i);

for (int t : p) {

isomorphic[i].push\_back(idx.at(t));

}

}

}

pattern(const pattern &p) = delete;

virtual ~pattern() {}

pattern &operator=(const pattern &p) = delete;

public:

/\*\*

\* estimate the value of a given board

\*/

virtual float estimate(const board &b) const {

// TODO

float value = 0;

for (int i = 0; i < iso\_last; i += 1) {

size\_t index = indexof(isomorphic[i], b);

value += operator[](index);

}

return value;

}

/\*\*

\* update the value of a given board, and return its updated value

\*/

virtual float update(const board &b, float u) {

// TODO

float u\_iso = u / iso\_last;

float value = 0;

for (int i = 0; i < iso\_last; i += 1) {

size\_t index = indexof(isomorphic[i], b);

operator[](index) += u\_iso;

value += operator[](index);

}

return value;

}

/\*\*

\* get the name of this feature

\*/

virtual std::string name() const {

return std::to\_string(isomorphic[0].size()) + "-tuple pattern " +

nameof(isomorphic[0]);

}

public:

/\*

\* set the isomorphic level of this pattern

\* 1: no isomorphic

\* 4: enable rotation

\* 8: enable rotation and reflection

\*/

void set\_isomorphic(int i = 8) { iso\_last = i; }

/\*\*

\* display the weight information of a given board

\*/

void dump(const board &b, std::ostream &out = info) const {

for (int i = 0; i < iso\_last; i++) {

out << "#" << i << ":" << nameof(isomorphic[i]) << "(";

size\_t index = indexof(isomorphic[i], b);

for (size\_t i = 0; i < isomorphic[i].size(); i++) {

out << std::hex << ((index >> (4 \* i)) & 0x0f);

}

out << std::dec << ") = " << operator[](index) << std::endl;

}

}

protected:

size\_t indexof(const std::vector<int> &patt, const board &b) const {

// TODO

size\_t index = 0;

for (size\_t i = 0; i < patt.size(); i += 1) {

index |= b.at(patt[i]) << (4 \* i);

}

return index;

}

std::string nameof(const std::vector<int> &patt) const {

std::stringstream ss;

ss << std::hex;

std::copy(patt.cbegin(), patt.cend(), std::ostream\_iterator<int>(ss, ""));

return ss.str();

}

std::array<std::vector<int>, 8> isomorphic;

int iso\_last;

};

/\*\*

\* before state and after state wrapper

\*/

class state {

public:

state(int opcode = -1)

: opcode(opcode), score(-1), esti(-std::numeric\_limits<float>::max()) {}

state(const board &b, int opcode = -1)

: opcode(opcode), score(-1), esti(-std::numeric\_limits<float>::max()) {

assign(b);

}

state(const state &st) = default;

state &operator=(const state &st) = default;

public:

board after\_state() const { return after; }

board before\_state() const { return before; }

float value() const { return esti; }

int reward() const { return score; }

int action() const { return opcode; }

void set\_before\_state(const board &b) { before = b; }

void set\_after\_state(const board &b) { after = b; }

void set\_value(float v) { esti = v; }

void set\_reward(int r) { score = r; }

void set\_action(int a) { opcode = a; }

public:

bool operator==(const state &s) const {

return (opcode == s.opcode) && (before == s.before) && (after == s.after) &&

(esti == s.esti) && (score == s.score);

}

bool operator<(const state &s) const {

if (before != s.before)

throw std::invalid\_argument("state::operator<");

return esti < s.esti;

}

bool operator!=(const state &s) const { return !(\*this == s); }

bool operator>(const state &s) const { return s < \*this; }

bool operator<=(const state &s) const { return !(s < \*this); }

bool operator>=(const state &s) const { return !(\*this < s); }

public:

/\*\*

\* assign a state (before state), then apply the action (defined in opcode)

\* return true if the action is valid for the given state

\*/

bool assign(const board &b) {

debug << "assign " << name() << std::endl << b;

after = before = b;

score = after.move(opcode);

esti = score;

return score != -1;

}

/\*\*

\* call this function after initialization (assign, set\_value, etc)

\*

\* the state is invalid if

\* estimated value becomes to NaN (wrong learning rate?)

\* invalid action (cause after == before or score == -1)

\*/

bool is\_valid() const {

if (std::isnan(esti)) {

error << "numeric exception" << std::endl;

std::exit(1);

}

return after != before && opcode != -1 && score != -1;

}

const char \*name() const {

static const char \*opname[4] = {"up", "right", "down", "left"};

return (opcode >= 0 && opcode < 4) ? opname[opcode] : "none";

}

friend std::ostream &operator<<(std::ostream &out, const state &st) {

out << "moving " << st.name() << ", reward = " << st.score;

if (st.is\_valid()) {

out << ", value = " << st.esti << std::endl << st.after;

} else {

out << " (invalid)" << std::endl;

}

return out;

}

private:

board before;

board after;

int opcode;

int score;

float esti;

};

class learning {

public:

learning() {}

~learning() {}

/\*\*

\* add a feature into tuple networks

\*

\* note that feats is std::vector<feature\*>,

\* therefore you need to keep all the instances somewhere

\*/

void add\_feature(feature \*feat) {

feats.push\_back(feat);

info << feat->name() << ", size = " << feat->size();

size\_t usage = feat->size() \* sizeof(float);

if (usage >= (1 << 30)) {

info << " (" << (usage >> 30) << "GB)";

} else if (usage >= (1 << 20)) {

info << " (" << (usage >> 20) << "MB)";

} else if (usage >= (1 << 10)) {

info << " (" << (usage >> 10) << "KB)";

}

info << std::endl;

}

/\*\*

\* accumulate the total value of given state

\*/

float estimate(const board &b) const {

debug << "estimate " << std::endl << b;

float value = 0;

for (feature \*feat : feats) {

value += feat->estimate(b);

}

return value;

}

/\*\*

\* update the value of given state and return its new value

\*/

float update(const board &b, float u) const {

debug << "update "

<< " (" << u << ")" << std::endl

<< b;

float u\_split = u / feats.size();

float value = 0;

for (feature \*feat : feats) {

value += feat->update(b, u\_split);

}

return value;

}

/\*\*

\* select a best move of a before state b

\*

\* return should be a state whose

\* before\_state() is b

\* after\_state() is b's best successor (after state)

\* action() is the best action

\* reward() is the reward of performing action()

\* value() is the estimated value of after\_state()

\*

\* you may simply return state() if no valid move

\*/

state select\_best\_move(const board &b) const {

state after[4] = {0, 1, 2, 3}; // up, right, down, left

state \*best = after;

for (state \*move = after; move != after + 4; move++) {

if (move->assign(b)) {

// TODO

move->set\_value(move->reward() + expect\_value(move->after\_state()));

if (move->value() > best->value())

best = move;

} else {

move->set\_value(-std::numeric\_limits<float>::max());

}

debug << "test " << \*move;

}

return \*best;

}

float expect\_value(const board &b) const {

float total = 0;

int cnt = 0;

for (int i = 0; i < 16; i++) {

if (!b.at(i)) {

board tmp(b);

tmp.set(i, 1);

total += 0.9 \* estimate(tmp);

tmp.set(i, 2);

total += 0.1 \* estimate(tmp);

cnt += 1;

}

}

return total / cnt;

}

/\*\*

\* update the tuple network by an episode

\*

\* path is the sequence of states in this episode,

\* the last entry in path (path.back()) is the final state

\*

\* for example, a 2048 games consists of

\* (initial) s0 --(a0,r0)--> s0' --(popup)--> s1 --(a1,r1)--> s1'

\* --(popup)--> s2 (terminal) where sx is before state, sx' is after state

\*

\* its path would be

\* { (s0,s0',a0,r0), (s1,s1',a1,r1), (s2,s2,x,-1) }

\* where (x,x,x,x) means (before state, after state, action, reward)

\*/

void update\_episode(std::vector<state> &path, float alpha = 0.1) const {

// TODO

float target = 0;

for (path.pop\_back(); path.size(); path.pop\_back()) {

state &move = path.back();

float error = target - estimate(move.before\_state());

target = move.reward() + update(move.before\_state(), alpha \* error);

}

}

/\*\*

\* update the statistic, and display the status once in 1000 episodes by

\* default

\*

\* the format would be

\* 1000 mean = 273901 max = 382324

\* 512 100% (0.3%)

\* 1024 99.7% (0.2%)

\* 2048 99.5% (1.1%)

\* 4096 98.4% (4.7%)

\* 8192 93.7% (22.4%)

\* 16384 71.3% (71.3%)

\*

\* where (let unit = 1000)

\* '1000': current iteration (games trained)

\* 'mean = 273901': the average score of last 1000 games is 273901

\* 'max = 382324': the maximum score of last 1000 games is 382324

\* '93.7%': 93.7% (937 games) reached 8192-tiles in last 1000 games (a.k.a.

\* win rate of 8192-tile) '22.4%': 22.4% (224 games) terminated with

\* 8192-tiles (the largest) in last 1000 games

\*/

void make\_statistic(size\_t n, const board &b, int score, int unit = 1000) {

scores.push\_back(score);

maxtile.push\_back(0);

for (int i = 0; i < 16; i++) {

maxtile.back() = std::max(maxtile.back(), b.at(i));

}

if (n % unit == 0) { // show the training process

if (scores.size() != size\_t(unit) || maxtile.size() != size\_t(unit)) {

error << "wrong statistic size for show statistics" << std::endl;

std::exit(2);

}

int sum = std::accumulate(scores.begin(), scores.end(), 0);

int max = \*std::max\_element(scores.begin(), scores.end());

int stat[16] = {0};

for (int i = 0; i < 16; i++) {

stat[i] = std::count(maxtile.begin(), maxtile.end(), i);

}

float mean = float(sum) / unit;

float coef = 100.0 / unit;

info << n;

info << "\t"

"mean = "

<< mean;

info << "\t"

"max = "

<< max;

info << std::endl;

for (int t = 1, c = 0; c < unit; c += stat[t++]) {

if (stat[t] == 0)

continue;

int accu = std::accumulate(stat + t, stat + 16, 0);

info << "\t" << ((1 << t) & -2u) << "\t" << (accu \* coef) << "%";

info << "\t(" << (stat[t] \* coef) << "%)" << std::endl;

}

scores.clear();

maxtile.clear();

}

}

/\*\*

\* display the weight information of a given board

\*/

void dump(const board &b, std::ostream &out = info) const {

out << b << "estimate = " << estimate(b) << std::endl;

for (feature \*feat : feats) {

out << feat->name() << std::endl;

feat->dump(b, out);

}

}

/\*\*

\* load the weight table from binary file

\* you need to define all the features (add\_feature(...)) before call this

\* function

\*/

void load(const std::string &path) {

std::ifstream in;

in.open(path.c\_str(), std::ios::in | std::ios::binary);

if (in.is\_open()) {

size\_t size;

in.read(reinterpret\_cast<char \*>(&size), sizeof(size));

if (size != feats.size()) {

error << "unexpected feature count: " << size << " (" << feats.size()

<< " is expected)" << std::endl;

std::exit(1);

}

for (feature \*feat : feats) {

in >> \*feat;

info << feat->name() << " is loaded from " << path << std::endl;

}

in.close();

}

}

/\*\*

\* save the weight table to binary file

\*/

void save(const std::string &path) {

std::ofstream out;

out.open(path.c\_str(), std::ios::out | std::ios::binary | std::ios::trunc);

if (out.is\_open()) {

size\_t size = feats.size();

out.write(reinterpret\_cast<char \*>(&size), sizeof(size));

for (feature \*feat : feats) {

out << \*feat;

info << feat->name() << " is saved to " << path << std::endl;

}

out.flush();

out.close();

}

}

private:

std::vector<feature \*> feats;

std::vector<int> scores;

std::vector<int> maxtile;

};

int main(int argc, const char \*argv[]) {

info << "TDL2048-Demo" << std::endl;

learning tdl;

// set the learning parameters

float alpha = 0.003125; // 0.1/32

size\_t total = 200000;

unsigned seed;

\_\_asm\_\_ \_\_volatile\_\_("rdtsc" : "=a"(seed));

info << "alpha = " << alpha << std::endl;

info << "total = " << total << std::endl;

info << "seed = " << seed << std::endl;

std::srand(seed);

// initialize the features

tdl.add\_feature(new pattern({0, 1, 2, 3, 4, 5}));

tdl.add\_feature(new pattern({4, 5, 6, 7, 8, 9}));

tdl.add\_feature(new pattern({0, 1, 2, 4, 5, 6}));

tdl.add\_feature(new pattern({4, 5, 6, 8, 9, 10}));

// restore the model from file

tdl.load("weight");

// train the model

std::vector<state> path;

path.reserve(20000);

for (size\_t n = 1; n <= total; n++) {

board b;

int score = 0;

// play an episode

debug << "begin episode" << std::endl;

b.init();

while (true) {

debug << "state" << std::endl << b;

state best = tdl.select\_best\_move(b);

path.push\_back(best);

if (best.is\_valid()) {

debug << "best " << best;

score += best.reward();

b = best.after\_state();

b.popup();

} else {

break;

}

}

debug << "end episode" << std::endl;

// std::ofstream out("scores.txt", std::ios::app);

// out << score << std::endl;

// out.close();

// update by TD(0)

tdl.update\_episode(path, alpha);

tdl.make\_statistic(n, b, score);

path.clear();

}

// store the model into file

tdl.save("weight");

return 0;

}