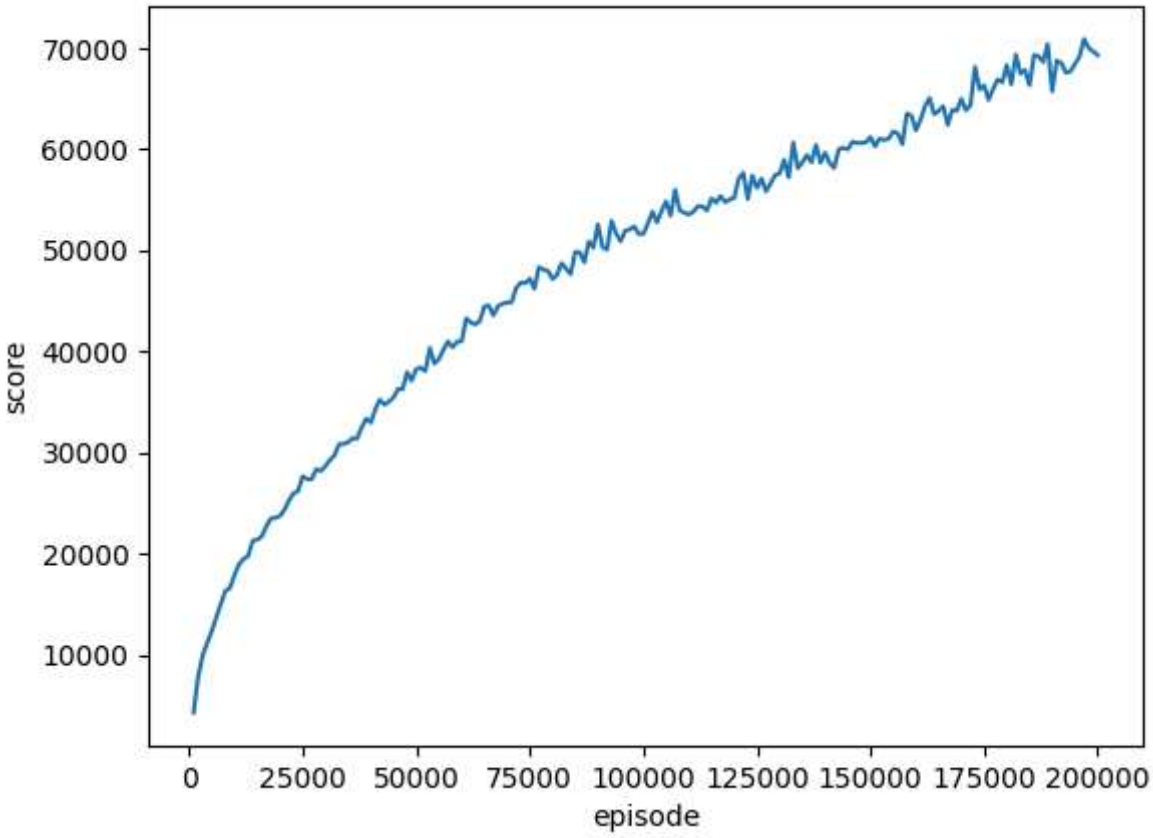
LAB1

A plot shows scores (mean) of at least 100k training episodes



Describe the implementation and the usage of n-tuple network.

因為 2048 的狀態總數非常巨大，所以需要 value function approximator 來估計盤面的 score，而 n-tuple network 在這次 LAB 就是作為這個角色。

n-tuple network 在 2048 的實作大致如下：

1. 取出幾個盤面位置形成一個 tuple
2. 以 tuple 為單位去建出一張表
3. 以盤面上對應位置的資訊去 encode 出 index，對表上該 index 進行查詢、更新

index of

- 分別取出盤面對應位置的值，並用左移將這些4-bit 的資訊 concat 成一個 mask (index)

```cpp
size_t indexof(const std::vector<int>& patt, const board& b) const {
    // TODO
    size_t index = 0;
    for (int t : patt) {
        index = (index << 4) | b.at(t);
    }
    return index;
}
```

estimate

- 給定一個盤面，對所有該 pattern 的同構查表，並累加回傳

```cpp
virtual float estimate(const board& b) const {
    // TODO
    float result = 0;
    for (int i = 0; i < iso_last; i++) {
        size_t index = indexof(isomorphic[i], b);
        result += operator[](index);
    }
    return result;
}
```

update

- 給定一個盤面和更新值，先將更新值除以同構數，對每個同構進行更新，並累加更新後的值回傳

```cpp
virtual float update(const board& b, float u) {
    // TODO
    float u_split = u / iso_last;
    float value = 0;
    for (int i = 0; i < iso_last; i++) {
        size_t index = indexof(isomorphic[i], b);
        operator[](index) += u_split;
        value += operator[](index);
    }
    return value;
}
```

Explain the mechanism of TD(0).

TD(0) 是 Temporal Difference Learning 的一種方法，特色是不用等一個完整的 episode 結束，就可以利用 S_t 和 S_{t+1} 進行更新。

詳細的更新公式如下：

$$V(S_t) \leftarrow V(S_t) + \alpha[r_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

Describe your implementation in detail including action selection and TD-backup diagram

next

- 我寫了一個 helper function，方便獲得一個當前 after-state popup 後可能的各個新盤面以及機率

```cpp
std::vector<std::pair<board, float>> next(const board& b) const {
    std::vector<std::pair<board, float>> next;
    std::vector<int> spaces;
    for (int i = 0; i < 16; i++) {
        if (b.at(i) == 0) {
            spaces.push_back(i);
        }
    }
    static std::vector<std::pair<int, float>> tiles = { {1, 0.9}, {2, 0.1} };
    for (auto &space : spaces) {
        for (auto [tile, prob] : tiles) {
            board next_board = b;
            next_board.set(space, tile);
            next.emplace_back(next_board, prob / spaces.size());
        }
    }
    return next;
}
```

select best move

- 利用 helper function 獲得所有可能的 state 以及機率後，分別 estimate 這些 state 並以機率作為權重，即可得到 $V(S_{t+1})$ 的期望值，選一個最大的回傳。

```cpp
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
            float value = 0;
            std::vector<std::pair<board, float>> all_possible_state = next(move->after_state());
            for (auto [next, prob] : all_possible_state) {
                float v = estimate(next);
                value += prob * v;
            }
            move->set_value(value);

            if (move->value() + move->reward() > best->value() + best->reward())
                best = move;
        } else {
            move->set_value(-std::numeric_limits<float>::max());
        }
        // debug << "test " << *move;
    }
    return *best;
}
```

update episode

- TD(0) 可以用相鄰兩步更新，從頭到尾跟從尾到頭更新理論上都可行，只要分別計算 $estimate(St), estimate(S_{t+1})$ 就能算出 error 並且 update。

- 如果從頭開始更新 $S_t \rightarrow S_{t+1} \rightarrow S_{t+2}...$。用 S_{t+1} 去更新 S_t，update weight 後要更新 S_{t+1} 時，需要再重算一次權重更新後的 S_{t+1}，造成計算的浪費。
- 因為 update 會順便回傳更新後的 $V(St)$，所以如果從尾巴來，更新 S_t 後可以存起來，順勢繼續往下更新 S_{t-1}，可以少計算一次。
- 這邊就是把上面的公式實做出來
  $V(S_t) \leftarrow V(S_t) + \alpha[r_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$

```cpp
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    // TODO
    path.pop_back();
    float target = 0;
    for (auto it = path.rbegin(); it != path.rend(); it++) {
        auto s = it;
        float error = target + s->reward() - estimate(s->before_state());
        target = update(s->before_state(), alpha * error);
    }
}
```