[Lab7] Temporal Difference Learning

0616014 楊政道

June 15, 2020

1 Report

1.1 A plot shows episode scores of at least 100,000 training episodes.

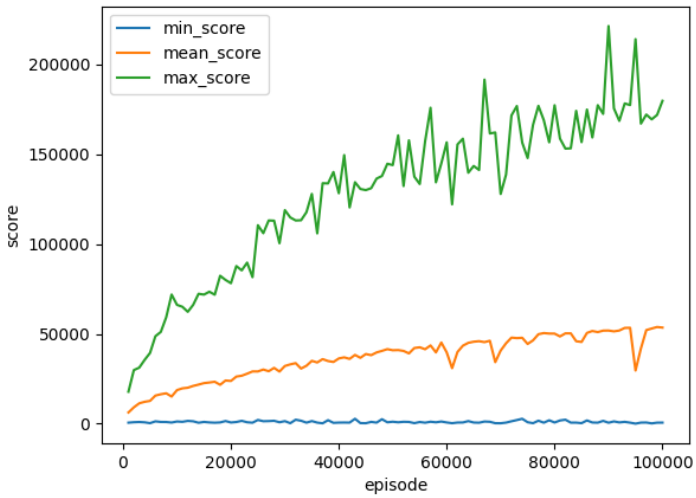

Figure 1: Performance

1.2 Describe your implementation in detail.

```
\subsubsection{pattern::estimate}
virtual float estimate(const board& b) const {
    // TODO
    float ret_value = 0;
    for (int i = 0 ; i < iso_last ; i++) {
        int idx = indexof(isomorphic[i], b);
        ret_value += operator[](idx);
    }
    return ret_value;
}
```

計算一個 pattern 在一個 board 上的數值總和, 輸入會給定一個 board, 要加上這個 pattern 所有同構方向的所有數值，並回傳回去。

1.2.1 pattern::update

```
virtual float update(const board& b, float u) {
    // TODO
    float u_mean = u / iso_last, ret_value = 0;
    for (int i = 0 ; i < iso_last ; i++) {
        int idx = indexof(isomorphic[i], b);
```

```
        operator[](idx) += u_mean;
        ret_value += operator[](idx);
    }
    return ret_value;
}
```

更新一個 pattern 的表的數值, 要找到所有和他同構的那些格子, 並且加上要更新的數值的平均。

1.2.2 pattern::indexOf

```
size_t indexof(const std::vector<int>& patt, const board& b) const {
    // TODO
    int ret = 0;
    for (int i = 0 ; i < (int)patt.size() ; i++)
        ret |= (b.at(patt[i]) << (4 * i));
    return ret;
}
```

給定某一個 pattern 的編號還有一個 board, 要根據 pattern 的編號去 board 中撈出數值並且拼起來, 就會得到 index。

1.2.3 learning::select_best_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() + estimate(move->after_state()));
            if (move->value() > best->value())
                best = move;
        } else {
            move->set_value(-std::numeric_limits<float>::max());
        }
        debug << "test " << *move;
    }
    return *best;
}
```

根據 TD learning 的更新公式, 這個 move 的 value 應該要是他這步得到的 reward 加上下一個 state(未加入新的方塊, after_state) 的估計值。

1.2.4 learning::update_epsisode

```
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    // TODO
    float sum = 0;
    path.pop_back();
    while (path.size()) {
        state& move = path.back();
        float loss = sum - (move.value() - move.reward());
        sum = move.reward() + update(move.after_state(), alpha * loss);
        path.pop_back();
    }
}
```

跑完一個 episode, 要根據這個 episode 中的決策路徑來更新 Q table, 其中 alpha 就是遞減的參數。

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

n-tuple 存在的理由為像是 2048 這種所有可能的數量太龐大, 無法每個 state 都開一格表格來存, 因此把整個版面切成幾個群組當成一個 pattern, 對於一個版面的估計值就是這些 pattern 所有同構的組合的

累積值。實作上會將版面上每個格子都編一個 16 進制的數字, 而一個 pattern 的組成即為幾個 16 進制的數字所組成的集合, 並事先預處理同構。

1.4 Explain the TD-backup diagram of V(state).

流程為有一個 state, 選擇了一個 action, 他會到一個 after-state, 並且 env 隨機放一個方塊到一個空位到達下一個 state, V(state) 的更新就是加上這裡的 TD error。

1.5 Explain the action selection of V(state) in a diagram.

action 的選擇為選擇他認為最好的狀態, 衡量一個狀態的好壞使用 n-tuple network。

1.6 Explain the TD-backup diagram of V(after-state).

流程為有一個 after-state, env 隨機放一個方塊到一個空位到達下一個 state, 接下來我選擇了一個 action, 他會到另一個 after-state, V(after-state) 的更新就是加上這裡的 TD error。

1.7 Explain the action selection of V(after-state) in a diagram.

action 的選擇為選擇他認為最好的狀態, 衡量一個狀態的好壞使用 n-tuple network。

1.8 Explain the mechanism of temporal difference learning.

TD learning 的機制為對於每個 action 都會做一次 Q 值參數更新, 不需要搜尋一段深度才有辦法更新, 更新的依據為現在的 state Q 值, 下一個 state 的 Q 值已經走這步的 reward。

1.9 Explain whether the TD-update perform bootstrapping.

Bootstrapping 為邊更新邊估計, TD learning 每次的更新的依據是下一個 state 的估計值, 因此 TD learning 有 bootstrapping。

1.10 Explain whether your training is on-policy or off-policy.

這是的 training 方式為 off-policy, 每次都會掃過所有的 action, 選擇最好的那個 action 來執行。

1.11 Other discussions or improvements.

在 action 選擇上可以多走幾步, 雖然會讓訓練速度變慢, 但是可以讓 agent 考慮的更完整, 可以讓 performance 有機會變高。

2 Performance

2.1 The 2048-tile win rate in 1000 games.

```
5356000 mean = 126248   max = 322412
        64      100%    (0.1%)
        128     99.9%   (0.3%)
        256     99.6%   (0.2%)
        512     99.4%   (0.7%)
        1024    98.7%   (2.7%)
        2048    96%     (12.7%)
        4096    83.3%   (19.1%)
        8192    64.2%   (55%)
        16384   9.2%    (9.2%)
```

Figure 2: Performance