Deep Learning and Practice

#Lab02 Temporal Difference Learning　309505002鄭紹文

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

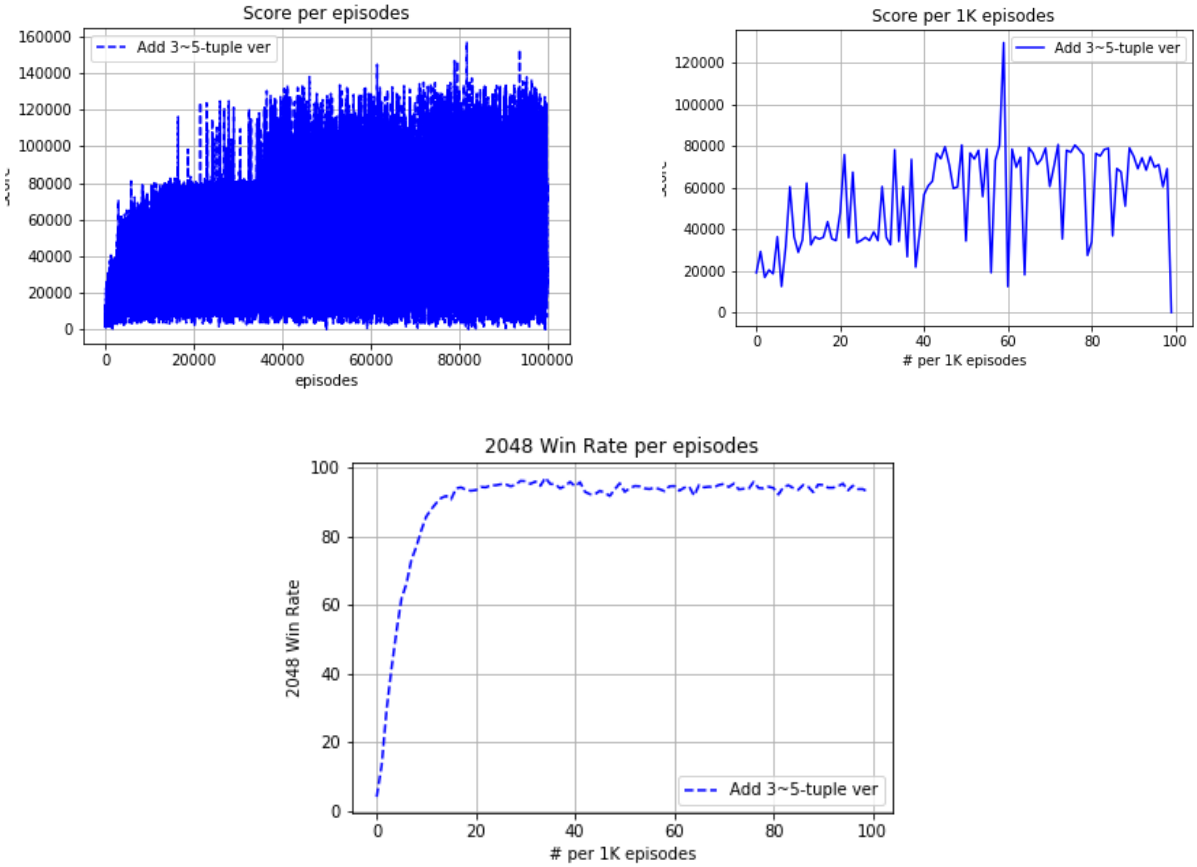

fig. 1

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

n-tuple network *(a.k.a. RAM-based neural network)*，tuple 可以定義成一個特徵所佔有的格數，用意在於當狀態有極多種可能時，我們肯定希望能夠全部存下來以方便做運算處理，然而建完的表格會過大而難以儲存，所以透過某些特徵代表目前的狀態，透過查出每個特徵的分數，加總後來代表盤面(以 2048 來說)做選擇。以 2048 為例，2048 的 board 版面為 4*4，若要紀錄所有 state 的各別的 V(s)，可行性不高，因 state 總數共有 16^16=2^64 個，若使用上 tuple 的概念，使用 4 組 6-tuple network，如此一來只會有 4 * 16^6 =2^26 種 state，相較下少了許多記憶體空間。我們知道當 tuple 數增加時，特徵數也會同時增加，產生更多的新圖形，然而由其他 paper 所做的實驗可知，6-tuple 為最大可構成的特徵圖案*(如 fig.2 所示)*，1-tuple 基本上沒有意義，2-tuple 幫助不大，所以可以從 3-tuple*(如 fig.2 所示)*, 4-tuple 開始去增加特徵數量，獲得較佳的結果。

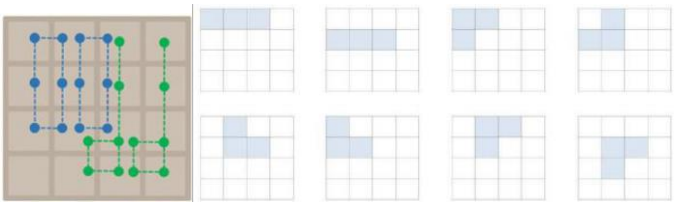

fig. 2

✓ Explain the mechanism of TD(0).

TD learning(Temporal-Difference learning)是 RL 中一個核心的算法，結合了 Monte Carlo 算法(MC)和動態規劃(DP)的想法，不僅可以直接從 sample 中學習，也可以如 DP 一樣使用 bootstrap 透過其他狀態值的估計更新當前狀態值。而 TD learning 可以將兩個狀態中所有的可能運算值做總和後，作為執行該步驟的 reward。

TD(0)的公式：

$$V(S_t) \leftarrow V(S_t) + \alpha[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{The TD target}} - V(S_t)]$$

跟 MC 的作法$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)$相比，最明顯差別就在於 MC 需等待一個 episode 結束才能更新，而 TD 可以不須等待整個 episode 跑完就可以透過 S t 與 S t+1 的差異來訓練。

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0, \forall s \in \mathcal{S}^+$)
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  until S is terminal

fig. 3

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

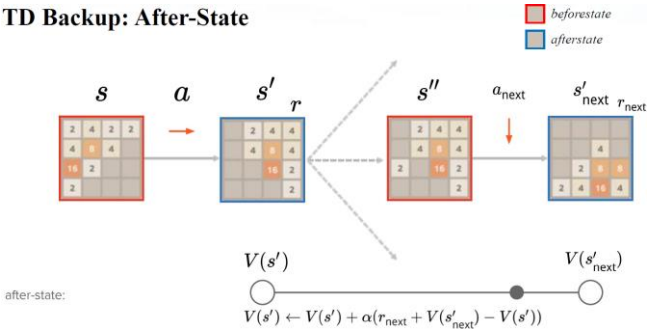

fig. 4

V(s')中的 s 代表的是經過某一 action(move)後，經過當前盤面環境隨機 pop out 一個 tile 的狀態。

TD-learning 時，利用剛剛 episode 所紀錄的結果算出 TD-traget: reward+V(s")來更新 V(s')。

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

選擇 action 時，因為 s 會有四種可能的 after states，所以必須先透過分別的分數 reward+V(s')去求出在哪一種 action 下會有最佳的選擇，所以四種都要跑一遍模擬再做選擇。

TD-after-state

function EVALUATE(s, a)
  $s', r \leftarrow$ COMPUTE AFTERSTATE(s, a)
  return $r + V(s')$

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

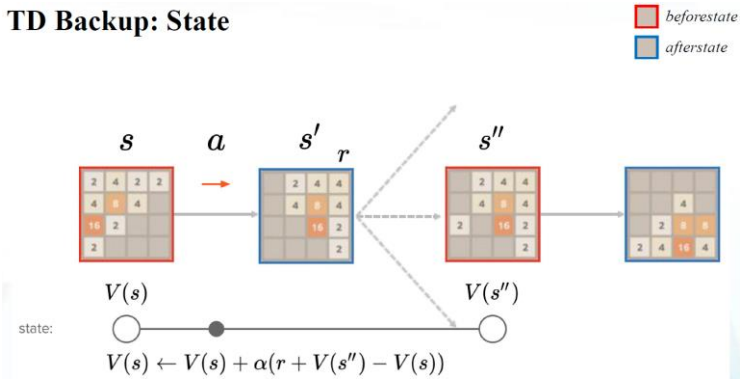

fig. 5

V(s)中的 s 代表的是在選擇做某個 action 之前的盤面狀態，再經過某 action 之後獲得 s'盤面，在此時去計算出在這個盤面後隨機 popup 出 tile 的所有可能性的總合(Sigma s")並存下來，以判斷說 S -> S'這個選擇是好還是不好。

TD-learning 時，利用紀錄的結果算出 TD-traget: reward+V(s")來更新 V(s)。

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

選擇 action 時，因為 s'->s"會有很多很多種可能性，我們必須要列出所有的可能性，並估計中每一可能性的值乘上機率$(\Sigma_{s'' \in s''} P(s,a,s'')V(S''))$，來判斷說這個 action 所對後面的影響到底是好還是不好。

TD-state

function EVALUATE(s, a)
  $s', r \leftarrow$ COMPUTE AFTERSTATE(s, a)
  $S'' \leftarrow$ ALL POSSIBLE NEXT STATES(s')
  return $r + \Sigma_{s'' \in S''} P(s,a,s'')V(s'')$

✓ Describe your implementation in detail.

此次 LAB 有 5 個 TODO 要填寫：

(1).

Pattern class 中的 estimate( ) function 計算當前 board 中的特定 pattern 組合與其 8 個 isomorphic Value(iso_last=8)之和，先得到其 index 之後透過 weight table 得到 weight 做加總。*這部分討論後認為跟 after-state 一樣*

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

fig. 6

(2).

Update function()是從 learning class 中的 update_episode() →learning class 中的 update()。update_episode()會丟入一個 board 以及 alpha*error 的參數，而 update()內可以看到一個 u_split，這個是做第一次 split，具體上是把 alpha*error 除以 feature 的個數，這部分可以想像成要更新的東西數值會平均分給那些 feature，以先分給每個 feature，然後每個 feature 再分給 8 個同購(for loop)，接著把各 feature 拆開代入 class pattern 中的 update()，透過 indexof()找 weight 的 table index，operator[](index) = operator[](index) + u_split 將取出來的 weight 加上剛剛兩段 split 後的數值做更新，最後再把更新的數值回傳。

這部分討論後認為跟 after-state 一樣

Training Arguments

- Learning rate: 0.1
  - Learning rate for features of n-tuple network with m features: $0.1 \div m$

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

fig. 7

(3).

在 estimate(), update() function 中我們都需要透過 index 找尋對應於 weight table 的 weight 值，而 indexof() function 就負責在輸入當前盤面以及 feature、isomorphic 時，得到並回傳所代表的 index，類似 weight table 的門牌號碼。舉例而言，若是 6-tuple network，因每個 tile 都用 4 個 bit 來表示，所以門牌號碼是 6*4bit=24bit，若如 fig.9 是 4-tuple network 則 4*4bit=16bit。

這部分討論後認為跟 after-state 一樣

```
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++)
        index |= b.at(patt[i]) << (4 * i);
    return index;
}
```

64	0	8	4
128	2 1		2
2	8 2		2
128	3		

0123	weight
0000	3.04
0001	−3.90
0002	−2.14
⋮	⋮
0010	5.89
⋮	⋮
0130	-2.01
⋮	⋮

fig. 8　　fig. 9

(4).

在 state 算法中，如同前面所述，需要計算出所有的可能性來判斷這個 action 到底好還是不好，故和 after state 寫法有極大不同。首先在先計算出在 action 之後的盤面到底有幾個空格，由於已知 2 出現的機率:4 出現的機率=9:1，換言之，要計算每個空格出現 2 以及 4 時的 V，再除以空格數及代表機率，這樣可以得到$\Sigma_{s'' \in s''} P(s,a,s'')V\left(S''\right)$，再加上 r，做比較之後可以得到選擇哪個 action 會是較佳的。

```
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
            // find after state 's space #//
            int space[16];
            int num = 0;
            for (int i = 0; i < 16; i++)
                if (move->after_state().at(i) == 0) {
                space[num++] = i;
            }
            float Sigma_value = 0;
            if(num){
                for(int i=num; i>0; i--){
                    board tmp_board;
                    tmp_board = move->after_state();
                    tmp_board.set(space[i-1], 1); //pop 2
                    // Sigma (Probability of all * 0.9 or 0.1 / num)
                    Sigma_value = Sigma_value + estimate(tmp_board)*0.9/num ;
                    tmp_board = move->after_state();
                    tmp_board.set(space[i-1], 2); //pop 4
                    Sigma_value = Sigma_value + estimate(tmp_board)*0.1/num;
                }
            }
            move->set_value(move->reward() + Sigma_value);
            if (move->value() > best->value())
                best = move;
        } else {
            move->set_value(-std::numeric_limits<float>::max());
        }
        debug << "test " << *move;
    }
    return *best;
}
```

TD-state

function EVALUATE(s, a)
　$s', r \leftarrow$ COMPUTE AFTERSTATE(s, a)
　$S'' \leftarrow$ ALL POSSIBLE NEXT STATES(s')
　return $r + \Sigma_{s'' \in S''} P(s, a, s'')V(s'')$

fig. 10

(5).

因為 path 紀錄了每一次的 s, s', a ,r, V(s)，分別代表了 before, after, opcode, reward, estimate value。首先要先丟掉最後一個紀錄，因為該步驟走得不好造成了遊戲結束，所以不學習它，然後在每一次讀取紀錄，由後往前做學習更新，code 的概念展示如 fig.12。

```
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    // TODO
    float exact = 0;
    for (path.pop_back() /* terminal state */; path.size(); path.pop_back()) {
        state& move = path.back();
        float error = exact - (estimate(move.before_state()) - move.reward());
        exact = update(move.before_state(), alpha * error);
    }
}
```

fig. 11

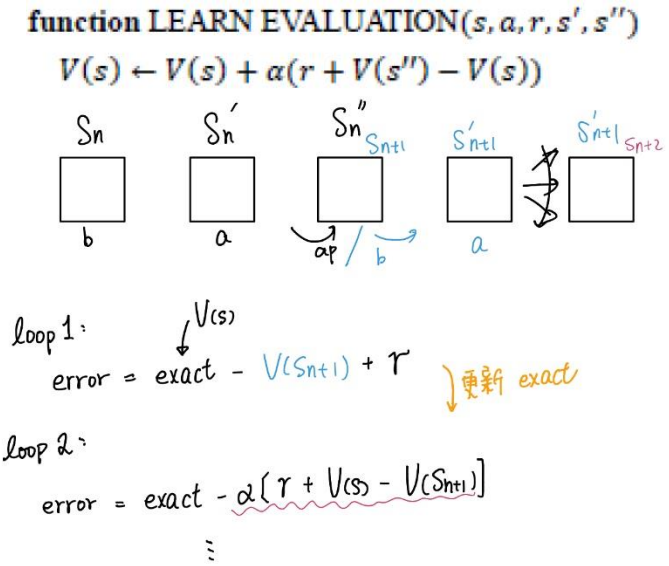

fig. 12

✓ Other discussions or improvements.

(1)

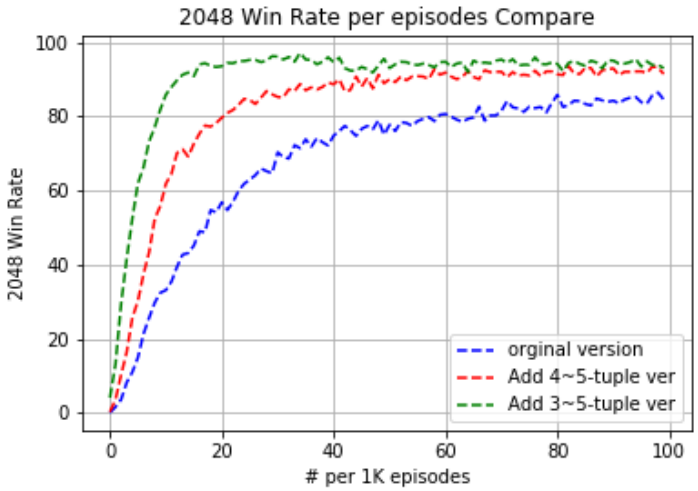

fig. 13

由前述可知，加上不同的 tuple，或者說增加其他不同種 feature(pattern)，增加更多學習的多樣性，會使結果來的更好，fig.13 中藍色虛線為最初的四組 pattern(6-tuple)的學習曲線，紅色虛線為加上 5-tuple 以及 4-tuple 的學習曲線，綠色虛線為 5-tuple、 4-tuple 以及 3-tuple 的學習曲線(每一千場記錄一次 score)，可以發現綠色虛線學習效果極高，不僅很早就達到較高的 score，同時也擁有最高的結果。

(2)

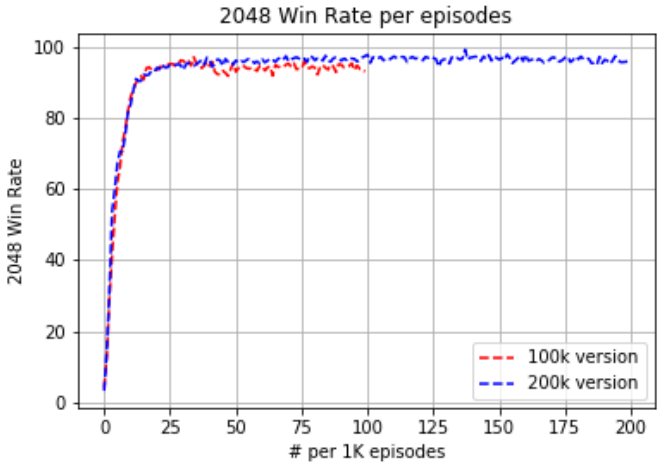

fig. 14

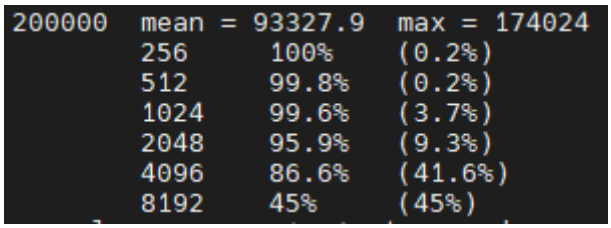

fig. 15

當多訓練 100K 場次會發現，最後的結果比原本得來的好，換言之，在 tuple 更改到一定時數量後，拉長訓練場次，有機會提高結果，由 fig.15 中發現 8192 亦提高到 45%。

(3).

version	第 9000 場時的 2048win rate	第 x 場時達到 50%	第 x 場時達到 80%	最後場次的 2048win rate	執行場數
Ver1：6-tuple*4	**29.7%**	**19K**	**60K**	**84.5%**	**100K**
Ver2： 6-tuple*7, 5-tuple*1 4-tuple*6	**54.7%**	**9K**	**23K**	**92.8%**	**100K**
Ver3： 6-tuple*7, 5-tuple*1 4-tuple*6, 3-tuple*6	**77%**	**5K**	**10K**	**96%**	**100K**

由表格可發現，增加 tuple 對提升速度以及表現都有很大的幫助。