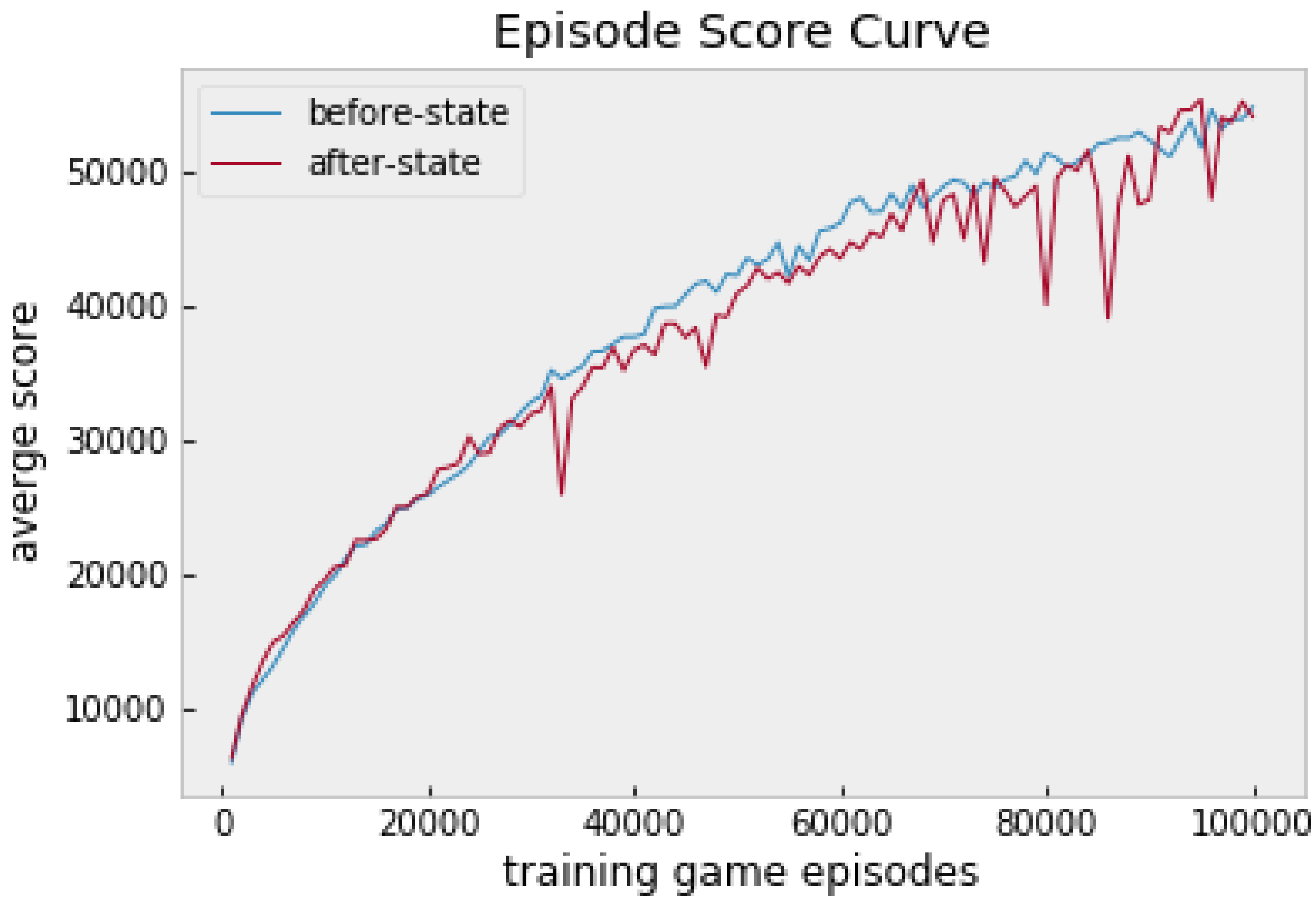
NCTU DLP Lab7-Report
Temporal Difference Learning

廖家鴻 0786009
2020/6/15

Outline

- Report (70%)
  - A plot shows episode scores of at least 100,000 training episodes (10%)
  - Describe your implementation in detail. (10%)
  - Describe the implementation and the usage of n-tuple network. (10%)
  - Explain the TD-backup diagram of V(state). (5%)
  - Explain the action selection of V(state) in a diagram. (5%)
  - Explain the TD-backup diagram of V(after-state). (5%)
  - Explain the action selection of V(after-state) in a diagram. (5%)
  - Explain the mechanism of temporal difference learning. (5%)
  - Explain whether the TD-update perform bootstrapping. (5%)
  - Explain whether your training is on-policy or off-policy. (5%)
  - Other discussions or improvements. (5%)
- Performance (30%)
  - The 2048-tile win rate in 1000 games, $\lceil \text{winrate}_{2048} \rceil$.

A Plot of Episode Scores with 100,000 Training Episodes



Implementation Details -- 1

從助教給的sample code中，可知這次implementation可以切分為5個class

1. Board：有處理「上 下 左 右」的moving functions，還有board status的處理。
2. Feature: 是class Pattern的virtual class，並且主要處理feature vector與weight table的計算
3. Pattern：有計算存取對應某tuple pattern的board value的index，並且每個pattern的8個isomorphism也是在此處理。
4. State: 處理每次game play中資訊，例如包括state transition、action taken、gained reward等等。
5. Learning: TD Learning的方法主要在此進行，包括決定best action的function、還有backward update function等等。

Implementation Details -- 2

```cpp
/**
 * estimate the value of a given board
 */
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;
}
```

左圖為第一個TODO，主要用來評估每種board型態下的board value。

先isomorphic中取出index值，然後算出board value值

```cpp
/**
 * update the value of a given board, and return its updated value
 */
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;
}
```

左圖為第二個TODO，主要用來更新每種board型態下的board value，並且return出這些更新值。

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

左圖為第三個TODO，主要用來取出index值

Implementation Details -- 3

```cpp
/*afterstate*/
state select_best_move_afterstate(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)) {
            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;
}
```

第四個TODO為選出最佳的V(S_t)跟action。
上圖為after-state的code，右圖則為before-state。
注意右圖before state的部分，需先計算從after-state到next state的期望值，因為下一狀態V未知。

```cpp
/*state*/
state select_best_move_state(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)) {
            board temp = move->after_state();
            board origin = temp;
            float plus2 = 0.0;
            float plus4 = 0.0;
            int count = 0;
            for (int i = 0; i < 16; i++){
                if (temp.at(i) == 0) {
                    temp.set(i, 1);
                    plus2 += estimate(temp);
                    temp = origin;
                    temp.set(i, 2);
                    plus4 += estimate(temp);
                    temp = origin;
                    count++;
                }
            }
            float exp = 0.9*(plus2/count) + 0.1*(plus4/count);

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

Implementation Details -- 4

```
/*afterstate*/
void update_episode_afterstate(std::vector<state>& path, float alpha = 0.1) const {
    float exact = 0;
    for (path.pop_back() /* terminal state */; path.size(); path.pop_back()) {
        state& move = path.back();
        float error = exact - (move.value() - move.reward());
        debug << "update error = " << error << " for after state" << std::endl << move.after_state();
        exact = move.reward() + update(move.after_state(), alpha * error);
    }
}
```

第五個TODO即為TD的model參數更新。

上圖為after-state的code，下圖則為before-state.

注意都是等每次episode結束後，從terminal update回initial。

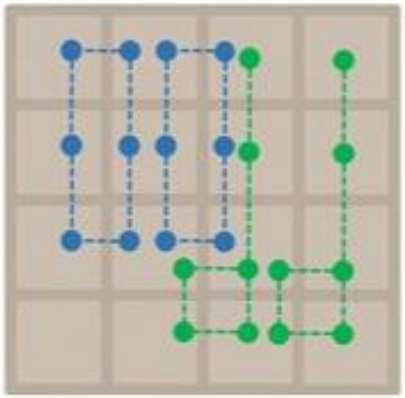
而TD的update function則是在”exact=...”這段code來處理。

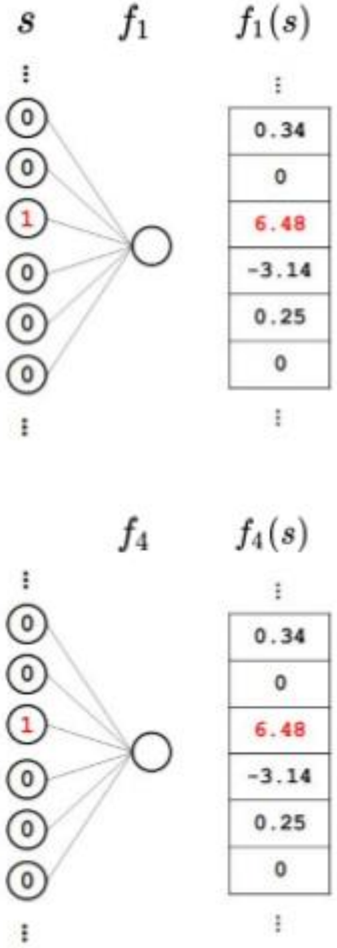
```
/*state*/
void update_episode_state(std::vector<state>& path, float alpha = 0.1) const {
    float exact = 0;
    for (path.pop_back() /* terminal state */; path.size(); path.pop_back()) {
        state& move = path.back();
        float error = move.reward() + exact - estimate(move.before_state());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

Implementation and the usage of n-tuple network -- 1

本次2048 Lab主要是用以下四個pattern的6-tuple，並且用這四個pattern的value值的總和來表示每個時間點的board的state value：

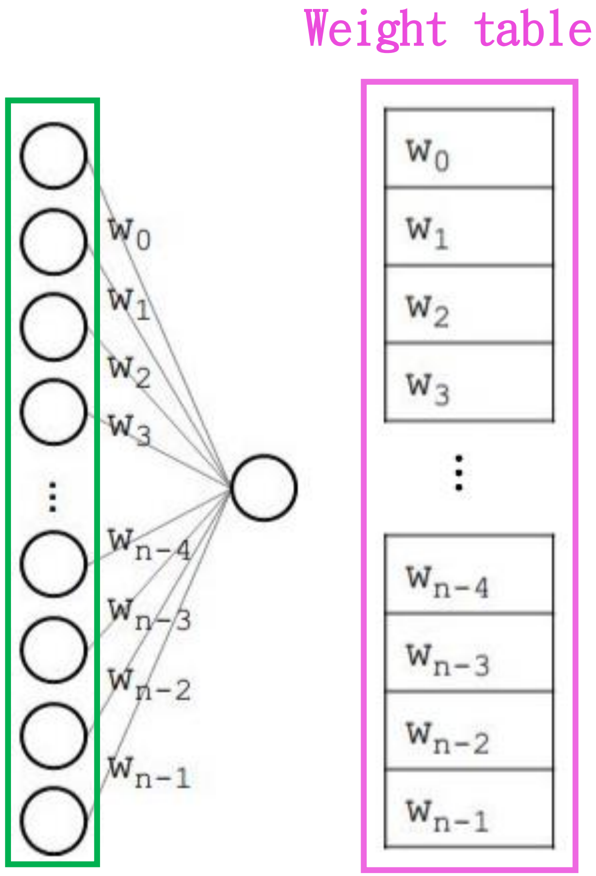
$$V(s) = f_1(s) + f_2(s) + f_3(s) + f_4(s)$$





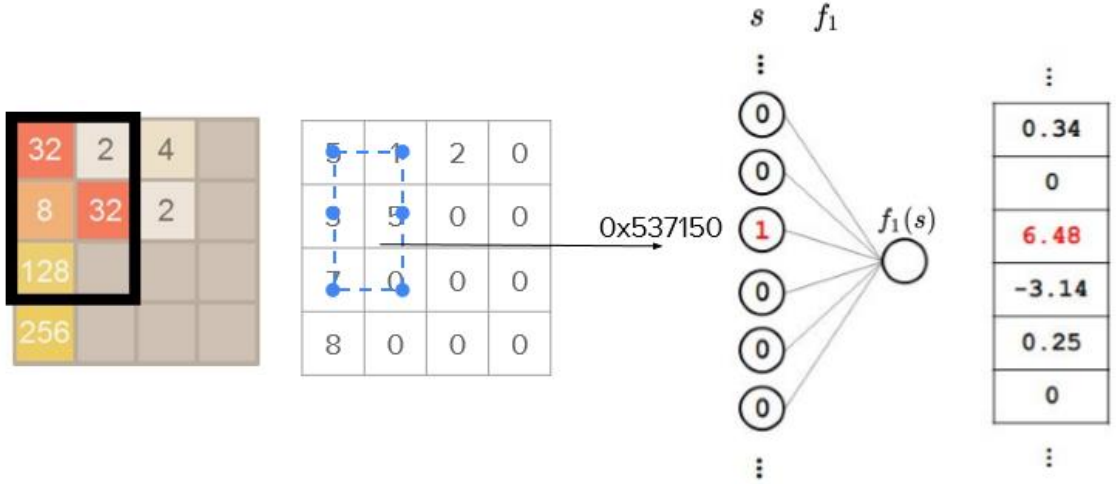
下圖是n-tuple network的示意圖，並且這裡的feature vector與weight table在code實作裡，是由「class feature」這個class來處理。

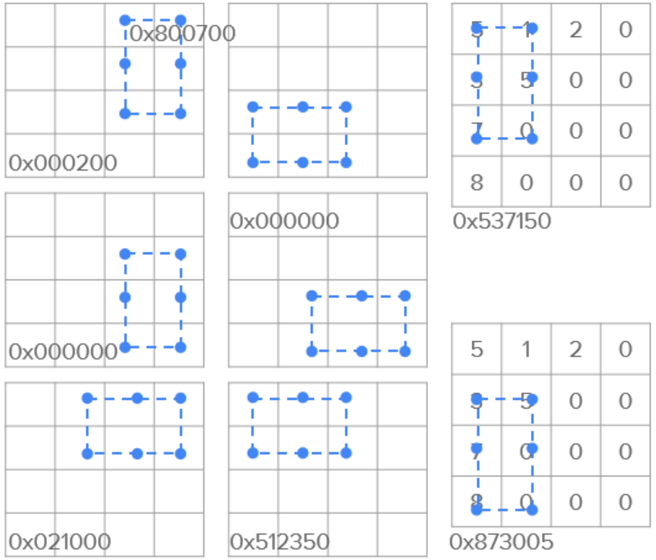
此為state feature，而其長度為8×16^6，並且會是共有8個1，其它都是0的sparse feature vector



Implementation and the usage of n-tuple network -- 2

下圖為第一種pattern的第一種結構的feature vector表示，一直到對應出weight table的值。





左圖為第一種pattern的8個同構(isomorphism)。因為同一pattern的各個結構的feature vector長度為16^6。而這4個pattern都各有8個同構，所以這4個pattern feature vector的長度為8×16^6。

下圖為「class pattern」裡，處理isomorphism的codes，並且4個pattern都是調用這裡的功能來處理各自的8個同構。

```
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;
```

Implementation and the usage of n-tuple network -- 3

N-tuple netwrok的value function，會用以下的formula來approximate：
(註：x(s)為feature vector，θ 為weight)

- Represent value function by a linear combination of features
  $$\hat{v}(S;\theta) = x(S)^{\mathrm{T}}\theta = \sum_{j=1}^{n} x_j(S)\theta_j$$
- Gradient of $\hat{v}(S,\theta)$:
  $$\nabla_\theta \hat{v}(S,\theta) = x(S)$$

6-tuple的四個pattern的total state數為4×16^6個
並且以下即為此approach法的weight table size：

```
6-tuple pattern 012345, size = 16777216 (64MB)
6-tuple pattern 456789, size = 16777216 (64MB)
6-tuple pattern 012456, size = 16777216 (64MB)
6-tuple pattern 45689a, size = 16777216 (64MB)
```

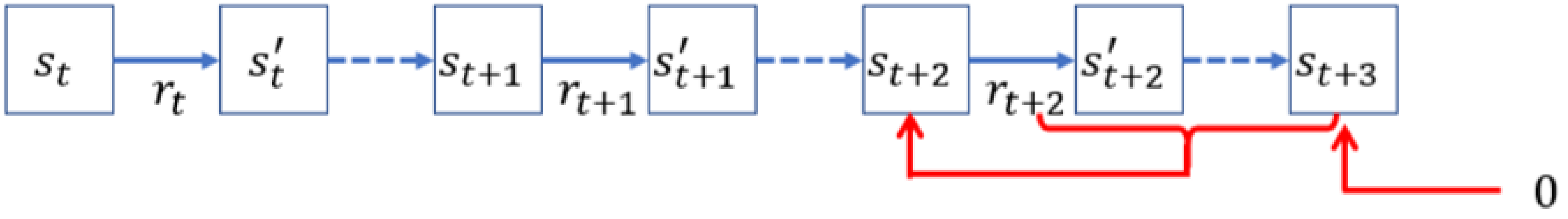
N-tuple network的objective function與weight update function如下所示：

- Objective function is to minimize the following loss in parameter θ. (note: $\hat{v}(S,\theta) = x(S)^{\mathrm{T}}\theta$)
  $$\mathcal{L}(\theta) = \mathbb{E}\left[\left(y_t - \hat{v}(S,\theta)\right)^2\right]$$
  $$\nabla_\theta \mathcal{L}(\theta) = \left(y_t - \hat{v}(S,\theta)\right) \cdot \nabla_\theta \hat{v}(S,\theta) = \Delta V \cdot x(S)$$
- Update features w: step-size * prediction error * feature value
  $$\theta \leftarrow \theta + \alpha \Delta V \cdot x(S)$$

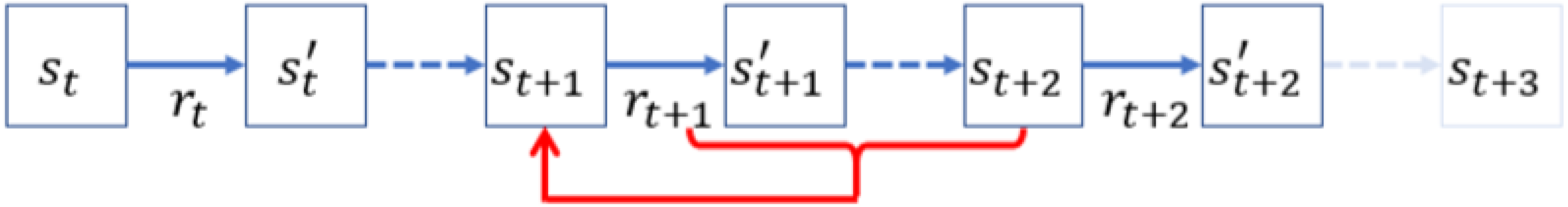
Table size 64MB的推算如下：
\# of entry = $16^6 \div 2^{20}$=16
64MB = 16M個entry*4byte的float

Explain the TD-backup diagram of V(state)

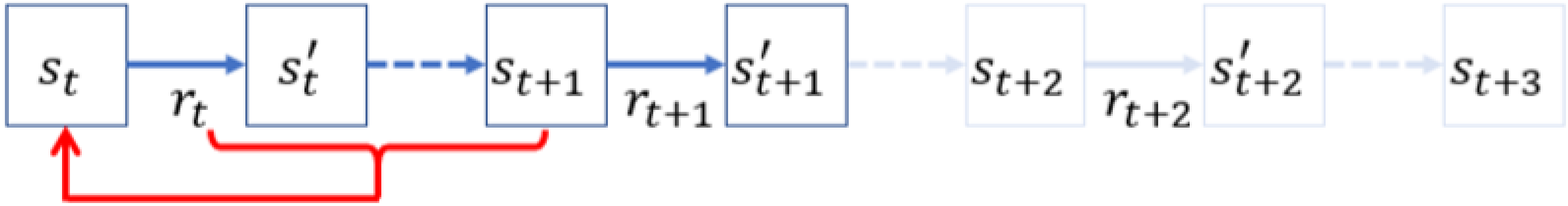
Step1. 假設某個episode遊戲terminal在(S_{t+3})，那就用(S_{t+3})與r_{t+2}來update (S_{t+2})



Step2. 接著再用(S_{t+2})與r_{t+1}來update (S_{t+1})

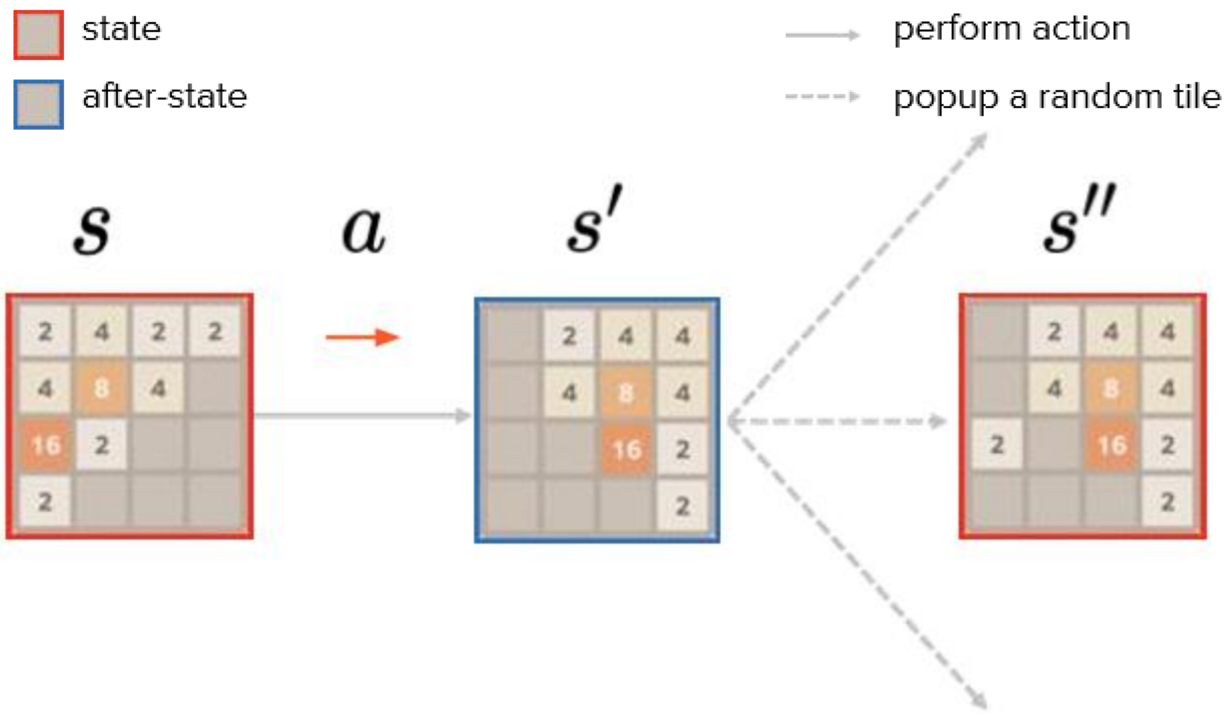


Step3. 以此類推來接連update回initial state，然後再進行下一個episode



註：(S′)為afterstate，而在此的TD update並不會用到afterstate

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



$$a \leftarrow \underset{a' \in A(s)}{\operatorname{argmax}} \ \text{EVALUATE}(s, a')$$

function EVALUATE(s, a)

$s', r \leftarrow$ COMPUTE AFTERSTATE(s, a)

$S'' \leftarrow$ ALL POSSIBLE NEXT STATES(s')

return $r + \Sigma_{s'' \in S''} P(s, a, s'') V(s'')$

由上圖可以看出，take不同的action (a)後，會前往不同的after-state (S') 。而在各個不同的after-state (S')之後，2與4這兩個tiles會以不同的機率在其中一個空格出現，所以會有多個可能的next-states (S'')。

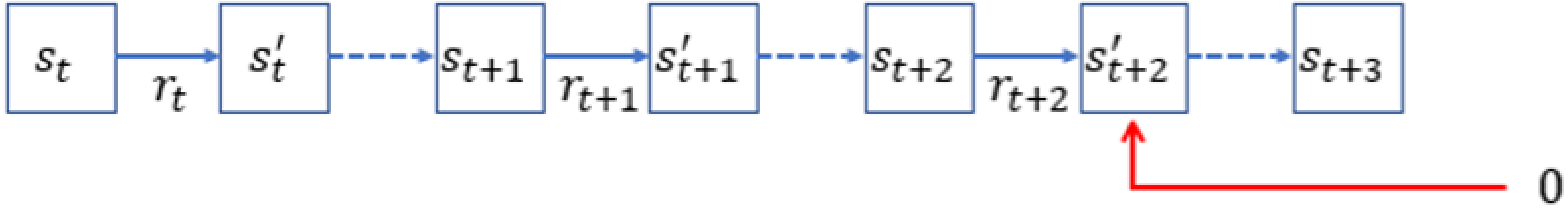
由上面的argmax可以知道，algorithm將take的action，是從EVALUATE這個function評估出來的。

而評估流程大致是，將當前的state與四個actions輸入到EVALUATE後，算出四組「reward加上next state的$V(S'')$期望值」，然後取其大者的action為policy。

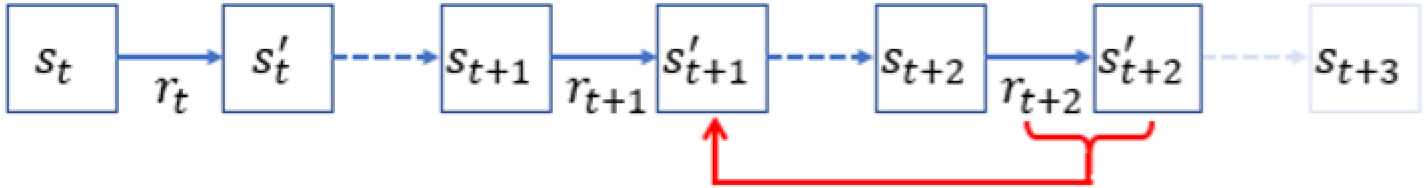
註：$V(S'')$對應的$P(s, a, s'')$與2與4這兩個tile random出現的機率有關。

Explain the TD-backup diagram of V(afterstate)

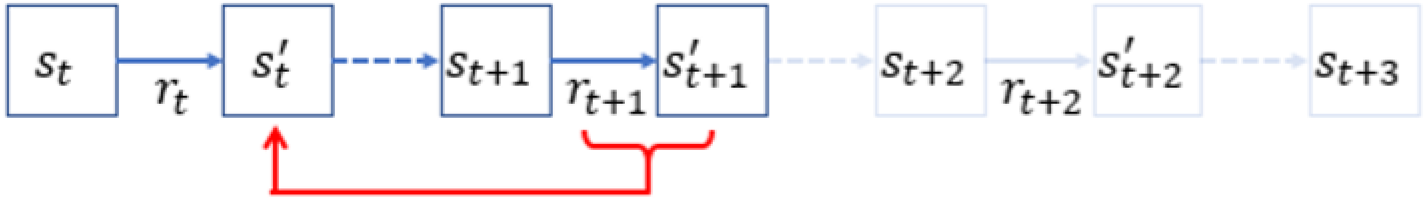
Step1. 假設某個episode遊戲terminal在(S_{t+3})，先update(S'_{t+2})



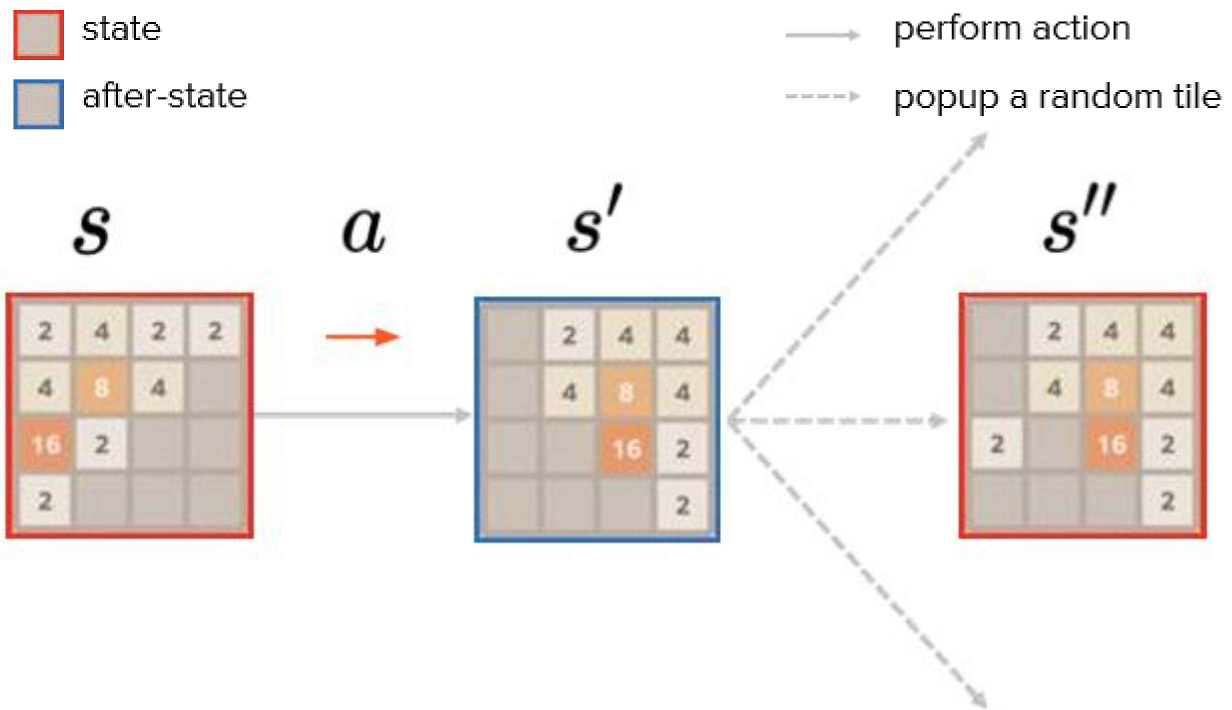
Step2. 接著再用(S'_{t+2})與r_{t+2}來update (S'_{t+1})



Step3. 以此類推來接連update回initial state，然後再進行下一個episode



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



$$a \leftarrow \underset{a' \in A(s)}{\operatorname{argmax}} \ \text{EVALUATE}(s, a')$$

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

由上圖可以看出，take不同的action (a)後，會前往不同的after-state (S') 。

由上面的argmax可以知道，algorithm將take的action，是從EVALUATE這個function評估出來的。

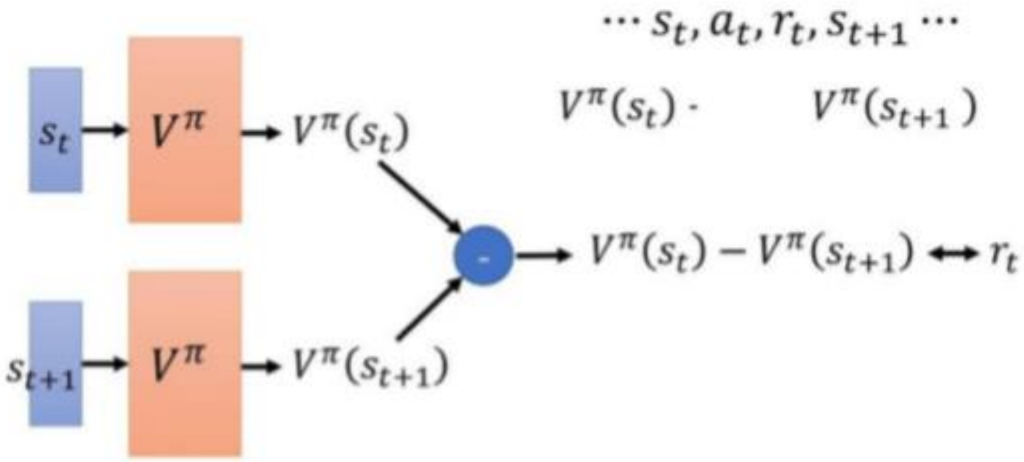
而評估流程大致是，將當前的state與四個actions輸入到EVALUATE後，算出四組「reward加上after state的$V(S')$值」，然後取其大者的action為policy。

註：這裡的EVALUATE function並不需考慮所有可能的next states。

Explain the Mechanism of TD Learning

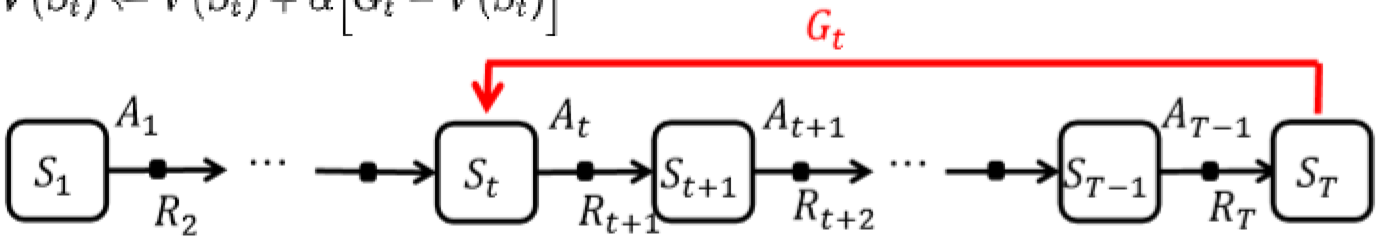
TD是用來更新V的機制。藉由兩個相鄰狀態的V值的差去計算接近轉換態的reward。來更新V使得V接近理想值。如圖所示：

- Temporal-difference approach



Monte-Carlo的update公式可以寫作：

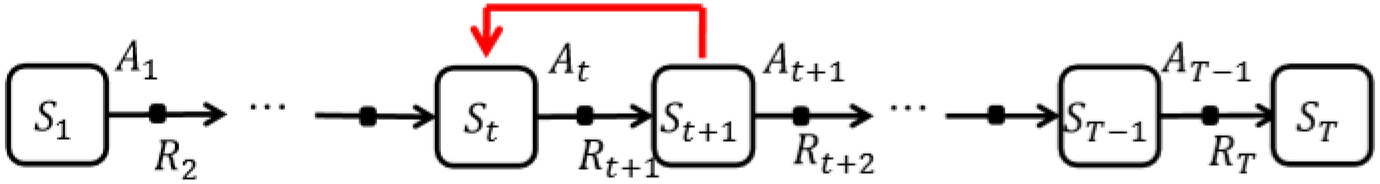
$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big]$$



其中Gt是時間t之後到該次終止時間的獎勵總和，α是學習率。但是TD無需等待到終止時間才進行更新，而是在下一步行動前就可以進行估計的更新了

TD Learning的update公式可以寫作：

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



註：雖然本次2048的lab是等episode terminal後才update回initial，但每次update step都是只看鄰近state的value與reward，所以還是跟Monte-Carlo不一樣。

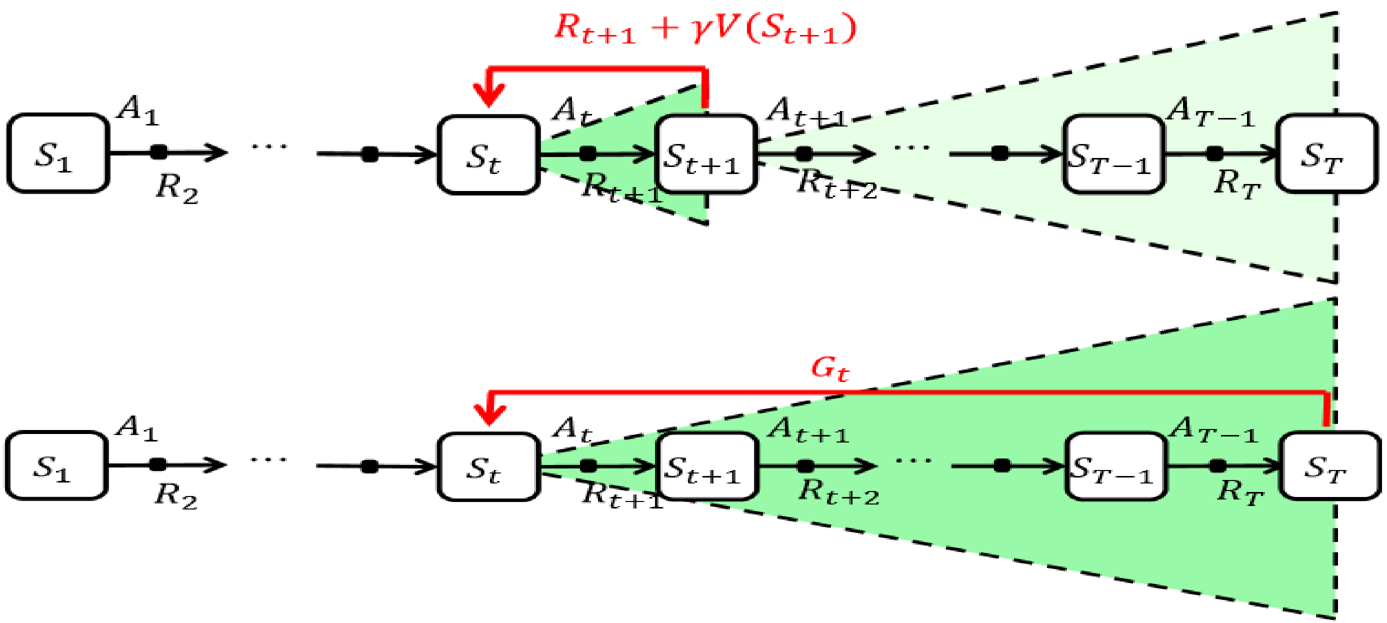
Explain whether the TD-update perform bootstrapping

TD Learning估算return的方式：通過採取action、對reward進行採樣，然後從當前對下一個狀態開始的return估算，來進行bootstrapping。

然後，它朝著此估計的方向邁出了一步，而不必等待整個獎勵序列。

對於bootstrapping的RL算法，是來自dynamic programming的想法。TD Learning有時被視為一種「中間算法」，它統一了Monte-Carlo Learning（對樣本reward進行採樣）和dynamic programming（對當前估計進行bootstrapping）。

Dynamic Programming的Value Iteration: $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s',r} p(s',r|s,a)(r + \gamma V(s'))$



TD Learning的update function:

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

Monte-Carlo Learning的update function:

$$V(S_t) \leftarrow V(S_t) + \alpha\left[G_t - V(S_t)\right]$$

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

首先回顧什麼是行為策略（Behavior Policy π(a|s)）和目標策略（Target Policy μ(a|s)）：
行為策略是agent與環境互動產生資料的策略，即在訓練過程中做決策；而目標策略在行為策略產生的資料中不斷學習、優化，即學習訓練完畢後拿去應用的策略。

而為瞭解決RL問題中的exploitation和exploration，我們可以用行為策略來保持探索性，提供多樣化的資料，而不斷的優化另一個策略（目標策略）。

On-policy的目標策略和行為策略是同一個策略。

Off-policy的目標策略和行為策略則是不同的，
行為策略可來自不同agent的經驗，或者是人類專家好的經驗(例如圍棋高手的policy)

所以從以上的探討中，會發現本次2048lab實作的algorithm雖然是episode terminal後才進行update，但仍是採取On-policy的方法。

TD(state)的update function => On-policy

function LEARN EVALUATION(s,a,r,s',s'')

$$V(s) \leftarrow V(s) + \alpha(r + V(s'') - V(s))$$

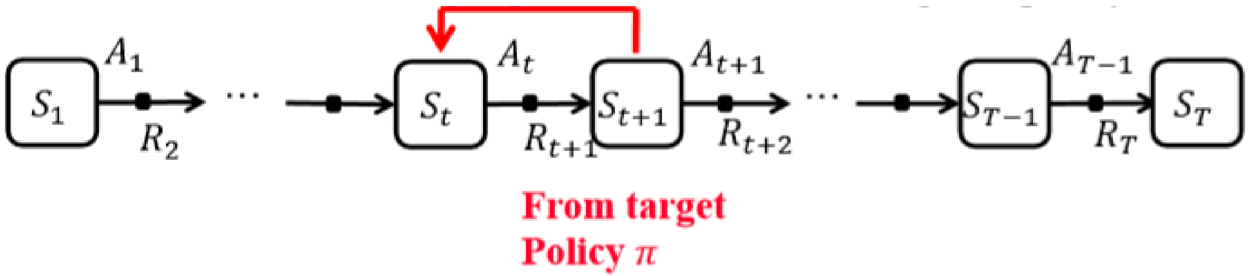
TD(after-state)的update function => On-policy

function LEARN EVALUATION(s,a,r,s',s'')

$$a_{next} \leftarrow \underset{a' \in A(s'')}{\operatorname{argmax}} EVALUATE(s'',a')$$

$$s'_{next}, r_{next} \leftarrow COMPUTE\ AFTERSTATE(s'',a_{next})$$

$$V(s') \leftarrow V(s') + \alpha(r_{next} + V(s'_{next}) - V(s'))$$



TD用importance sampling對V(S)做Off-policy時需採用的update function：

$$V(S_t) \leftarrow V(S_t) + \alpha\left(\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}(R_{t+1}+\gamma V(S_{t+1})) - V(S_t)\right)$$

Other Discussions or Possible Improvement Method

On-policy的目標策略π(a|s)和行為策略μ(a|s)是同一個策略，其好處就是直接利用agent當下附近的資料就可以優化其策略，然而這樣的處理常會導致該策略其實只是在學習一個局部最優，因為On-policy的策略沒辦法很好的同時保持即exploration又exploitation。

所以其中一個可能可以improve的方向是改用Off-policy的方法。

而Off-policy將目標策略和行為策略分開，可以在保持探索的同時，更能求到全域最優值。但其難點在於：如何在一個策略下產生的資料來優化另外一個策略？通常都會利用到“Important Sampling”这种技巧來解决，而TD Learning用importance sampling對V(S)做Off-policy時需採用的update function改寫如下：

$$V(S_t) \leftarrow V(S_t) + \alpha\left(\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}(R_{t+1}+\gamma V(S_{t+1})) - V(S_t)\right)$$

然而上式中，假若有π(a|s)值很大、μ(a|s)極小的情況發生的話，將會導致update很不穩定。所以進一步想，原來後來有人用Q-learning來解2048這個遊戲，其實可以避開上述importance sampling方法對TD target乘上那ratio值的不穩定問題。然後用Q-learning的話還有一個好處是也可以巧妙避開從after-state到next state的transition probabilities的問題，減少很多計算量。

不過我太晚才搞懂吳老師這個上課時有提到的事，也就是用Q-learning有機會改善on-policy的local minimum問題以及importance sampling的不穩定問題，所以來不及嘗試，以上算是把心得與討論寫下來。

The 2048-tile win rate in 1000 games, [$winrate_{2048}$]

以下是after-state的結果

```
979000  mean = 103382   max = 292704
    64  100%     (0.1%)
    128 99.9%    (0.2%)
    512 99.7%    (0.7%)
    1024    99% (5.8%)
    2048    93.2%    (16.8%)
    4096    76.4%    (26.4%)
    8192    50% (46.4%)
    16384   3.6%     (3.6%)
```

以下是after-state再試著多train的結果

```
1293000 mean = 116039   max = 333232
        64      100%    (0.2%)
        128     99.8%   (0.2%)
        256     99.6%   (0.1%)
        512     99.5%   (0.9%)
        1024    98.6%   (5%)
        2048    93.6%   (15.1%)
        4096    78.5%   (21.3%)
        8192    57.2%   (49.8%)
        16384   7.4%    (7.4%)
```

以下是before-state的結果

```
992000  mean = 105550   max = 225852
        128     100%    (0.2%)
        256     99.8%   (0.3%)
        512     99.5%   (1.1%)
        1024    98.4%   (4.1%)
        2048    94.3%   (9%)
        4096    85.3%   (30.9%)
        8192    54.4%   (54.2%)
        16384   0.2%    (0.2%)
```

另外有發現，有出現16384的win rate最高者
，為左下圖的結果。

The End