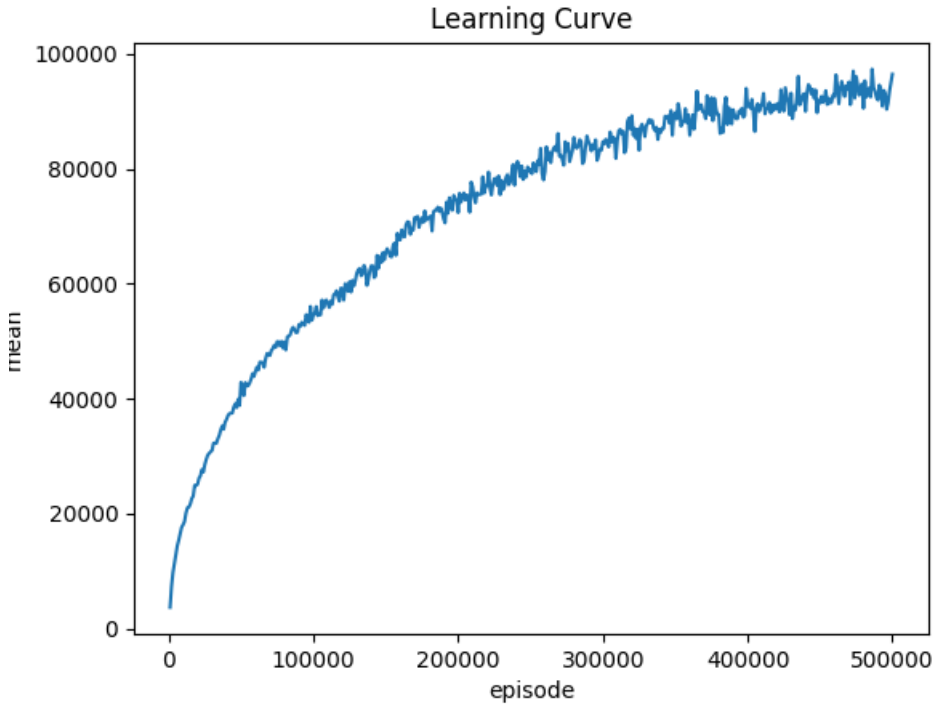
1. A plot shows scores (mean) of at least 100k training episodes
   這是從訓練至 500000 個 episode 後所印出的 training 過程的 mean。



2. Describe the implementation and the usage of n-tuple network
   2048 的盤面總共有 16 格，而每一格都有 17 種可能(空格、2 的一次、2 的二次、…、2 的十六次)，因此所有可能的盤面總共有 17 的 16 次個！如果要計算每一種盤面的 value 估計值，我們可能就需要開一個大小為 17 的 16 次的表格來儲存，但這不只會使記憶體用量無法負荷，更重要的是玩遊戲的過程中大多只會不斷遇到「從未見過」的盤面，那些曾經見過、更新過的盤面幾乎再也不會遇到，這樣子會讓訓練變得很困難。
   不過，對於 2048 遊戲的盤面來說，可以藉由學習特徵來讓相似的盤面能夠被預測出類似的 value 值。經由前人的研究結果發現，對於 2048 遊戲來說，光是盤面上取一小塊，就能成為很有用的特徵。因此，我們可以選取一塊位置作為盤面的「feature」，而計算 value 值時也只會對盤面上的那一小塊 feature 做操作。透過 n-tuple network(n 指的是取的 feature 中有幾塊格子)，我們就可以將 network 中的每個 weight，作為一個 feature 值的 value 估計值，並在遊戲中不斷更新，使得最後盤面 value 估計值可以趨近真實的 value。

   同時，在 2048 遊戲中，有時會出現雖然長得不一樣，但其實完全等價的盤面(像是把一個盤面順時針轉 90 度後出現的盤面，和原本的盤面就是等價的)。因此，在計算盤面的估計值時，應該要考慮 feature 的 4 種旋轉*2 種鏡像=總共 8 種的 isomorphism，並將這 8 個 isomorphism 的 value 估計值加起來，作

為此盤面的此 feature 的 value 估計值。當盤面上有多個 feature 時，就要將各個 feature 的 value 估計值都加起來，才能作為此盤面的 value 估計值。

3. Explain the mechanism of TD(0)

   比起 monte-carlo learning 認為所有真實 reward 的和才是 value function 的學習目標；TD(0)相信目前的 value function 所估計的下一個 state 的 value 值就已經是對的，所以只要再加上這一步得到的 reward，就應該是真正這個 state 能得到的 value，因此學習目標為 R(*t+1)* + V(S*(t+1)*)。因此在訓練中，TD(0)就會去多看下一步，看看自己現在的 value 估計值還和這個學習目標差多少，然後就以此差距*step size 去更新目前的 V(St)。

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

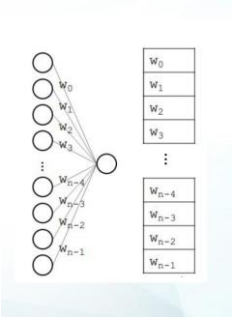
   首先說明在 n-tuple network 中提到的估計各個盤面 value 的方法：

   下面的程式是 sample code 中有提供的，它會計算出一個給定的 feature，它的其他 isomorphism(pattern)是對應到盤面那些 index。

```
451         for (int i = 0; i < 8; i++) {
452             board idx = 0xfedcba9876543210ull;
453             if (i >= 4) idx.mirror();
454             idx.rotate(i);
455             for (int t : p) {
456                 isomorphic[i].push_back(idx.at(t));
457             }
458         }
459     }
```

然後在 function “indexof” 中，會去計算一個給定的 pattern，是在 weight 的第幾個 index 中儲存這個 pattern value。這個 index 是利用 feature 在 board 上的值以 16 進位來表示的，所以只要將 feature 中的值轉回成 10 進位即可找到此 index。

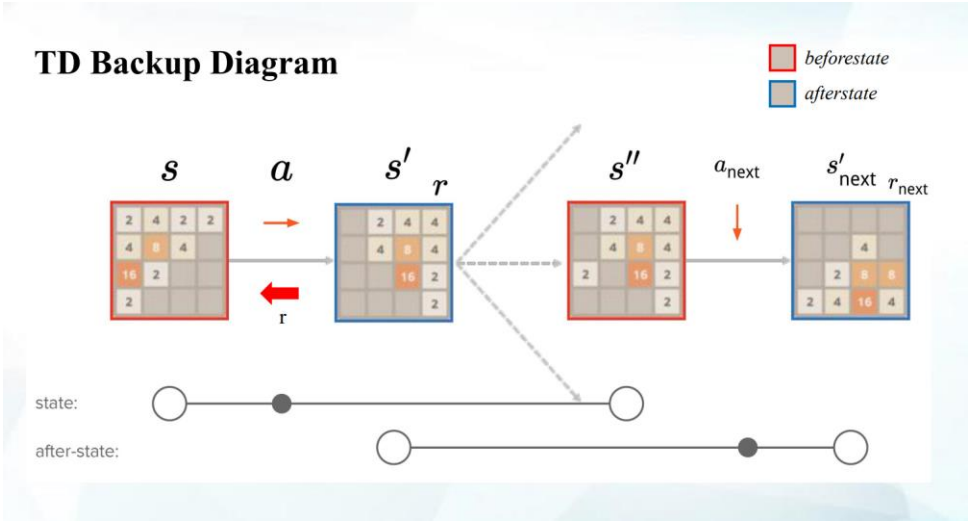
```
532     size_t indexof(const std::vector<int>& patt, const board& b) const {
533         // TODO
534         // 算出feature中weight為1的位置
535         size_t index_of_weight = 0;
536         int mul = 1;
537         for (int i = patt.size() - 1; i >= 0; i--) {
538             int board_value = b.at(patt[i]);
539             index_of_weight += board_value * mul;
540             mul *= 16;
541         }
542
543         return index_of_weight;
544     }
```



然後在 function “estimate” 中，就可以透過取出代表各個 isomorphism pattern 的 weight value，並將它們相加來得到此盤面的此 feature 的 value 估計值。

```
469     virtual float estimate(const board& b) const {
470         // TODO
471         float value = 0;
472         for (int i = 0; i < iso_last; i++){
473             size_t id_of_weight =  indexof(isomorphic[i], b);
474             value += weight[id_of_weight];
475         }
476 
477         return value;
478     }
479 
```

接著在訓練過程中，如同以下的 TD Backup Diagram 以及前述，這次的 2048 遊戲中 N-tuple network 的 weight 儲存了 before state 的 estimated value。每當要更新 weight value 時，會透過去選擇一個目前最好的 action(等等會說明如何做的)，做了 action 後就會變成 after state，並得到做了這個 action 的 reward。此 after state 要變成下一個 before state 時，因為 popup 出的新數字不論是位置還是數值都有隨機性，因此就會像圖中所示，after state 之後是有 branch 的。



在一個遊戲 episode 結束之後，我們就得到了整個遊戲過程中發生的確定的 before state 以及 after state，此時我們就能從 episode 的最末端開始一步步更新 value 的估計值，更新方式如同前面 TD(0)的說明所述。

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

在下面的 function “update_episode”中，我從 episode 的倒數第二個盤面開始計算要更新的 value(因為最後一個盤面就是死掉了，因此預期的 value 就是 0，所以不用更新)。每個 before state value 的學習目標就是下一個 state 的 estimated value + 這個 before state 做了 action 後得到的 reward。減掉目前 state 的 value 後得到兩者的差距 error 後再乘上 stepsize alpha 後就是 weight 們要更新的總和。

```cpp
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    // TODO
    float true_next_state_value = 0;
    for(int i = path.size()-2; i>=0; i--){
        state& this_state = path[i];
        float error = (true_next_state_value + this_state.reward()) - this_state.value();

        // prepare for the next iteration
        true_next_state_value = this_state.reward() + update(this_state.before_state(), alpha * error);

    }
}
```

所以要更新的總值首先會被傳入 class learning 中的 update，更新總值會平均分配給所有的 feature，然後呼叫 class pattern 中的 update 做 weight 的更新。

```cpp
689         float update(const board& b, float u) const {
690             debug << "update " << " (" << u << ")" << std::endl << b;
691             float u_split = u / feats.size();
692             float value = 0;
693             for (feature* feat : feats) {
694                 value += feat->update(b, u_split);
695             }
696             return value;
697         }
```

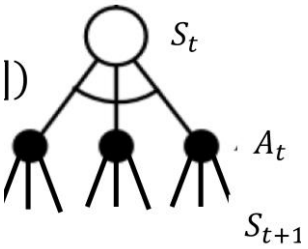
在 class pattern 中的 update 中，會再將傳進來的更新值再平均分配給所有 isomorphism。接著透過 indexof 找到要更新的 weight，然後把 weight 加上這個要更新的值。

```cpp
/**
 * update the value of a given board, and return its updated value
 */
virtual float update(const board& b, float u) {
    // TODO
    // u -> sum of updated value, should be distributed to those weights needed to be updated (# = iso_last)
    float u_split = u / iso_last;
    float value = 0;
    for (int i = 0; i < iso_last; i++){
        size_t id_of_weight =  indexof(isomorphic[i], b);
        weight[id_of_weight] += u_split;
        value += weight[id_of_weight];
    }

    return value;


}
```

接著說明遊戲中是如何選擇最好的 action。每個 before state 都可能有多個可以選擇的 action，而做了每一個 action 後，popup 出的新數字在位置和數值都有隨機性，因此變成下一個 before state 前，畫像下圖一樣有很多的分支。



要從中選擇最好的 action 的方式，就是假裝去做每個 action，到了 after state

後，就去模擬所有可能的下一個 before state 的情況，把這些各個 before state 的估計 value 用根據出現的機率加總起來，然後去比較做了各個 action 的估計 value，能產生最大 value 的那個 action，就是我們要選擇的 action。

下圖中就是去 assign 各種 action 給目前的 before state，如果可以 assign 給它的話(能做這個 action)，那就會去看盤上所有空格，然後在那些位置擺上 4(機率 0.1)或 2(機率 0.9)，計算新盤面的估計值，並與 reward 做相加。744 行就是去比較做了此 action 後可能得到的 value 是不是比目前認為最好的還要大，如果比較大的話，就把它紀錄起來。所有 action 都試過之後，存起來的 action 就是要選的最好的 action。

```
709     state select_best_move(const board& b) const {
710         state after[4] = { 0, 1, 2, 3 }; // up, right, down, left
711         state* best = after;
712         float best_predicted_value = -std::numeric_limits<float>::max();
713         for (state* move = after; move != after + 4; move++) {
714             if (move->assign(b)) {
715                 // TODO
716                 // move->before becomes b;
717                 // move->after becomes b' after the ith actoin has been applied
718                 // move->esti stores the reward
719                 // all of them are private
720                 // -> compute the predicted value considering all popup probability
721                 int empty_in_after_board = 0;
722                 float predicted_value = 0;          // predicted value = Rt+1  +  V(St+1)
723                 for(int i = 0; i < 16; i++){
724 
725                     if(move->after_state().at(i)==0){
726                         // empty tile
727                         board next_before_board = board(move->after_state());
728 
729                         // popup "2" in the empty tile and calculate the predicted value of this before state;
730                         next_before_board.set(i, 1);
731                         predicted_value += 0.9 * estimate(next_before_board);
732                         // popup "4" in the empty tile and calculate the predicted value of this before state;
733                         next_before_board.set(i, 2);
734                         predicted_value += 0.1 * estimate(next_before_board);
735 
736                         empty_in_after_board++;
737                         // empty_in_after_board.push_back(i);
738                     }
739                 }
740                 predicted_value /= empty_in_after_board;
741                 predicted_value += move->reward();
742 
743                 move->set_value(estimate(move->before_state()));
744                 if (predicted_value > best_predicted_value){
745                     best = move;
746                     best_predicted_value = predicted_value;
747                 }
748 
```

Reference: https://ko19951231.github.io/2021/01/01/2048/