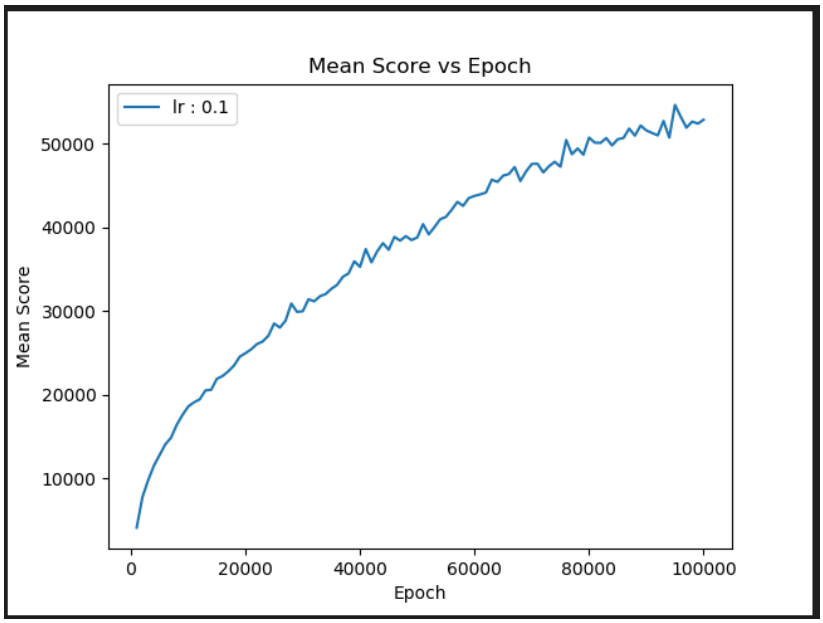
Deep Learning

Lab3: Temporal Difference Learning

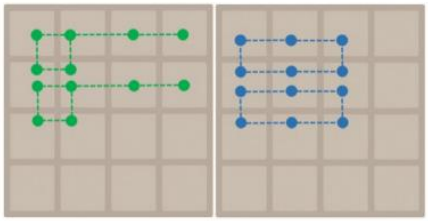
311512064 鄧書桓

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



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

n-tuple 為網格中的特徵提取器，通常 tuple 為相鄰的像素組成，並在學習過程中根據 agent 的 action 與 reward 來更新特定位置 tuple 的權重，以此來獲得最佳的 policy。簡單來說，就是將像素值(這邊為棋盤格)轉換為特徵向量，從而學習到最優的策略。



```
// initialize the features
tdl.add_feature(new pattern({0, 1, 2, 3, 4, 5}));
tdl.add_feature(new pattern({4, 5, 6, 7, 8, 9}));
tdl.add_feature(new pattern({0, 1, 2, 4, 5, 6}));
tdl.add_feature(new pattern({4, 5, 6, 8, 9, 10}));
```

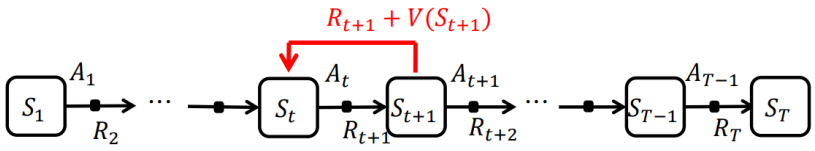
本實驗是使用 4 個 6tuples 來實現，示意圖如上左圖所示，原本的 sample code 就有給定好了，如上右圖所示，我有嘗試做些更動，但訓練出來的值並沒有更好。

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

另外，如上圖所示，這邊是使用 isomorphic 的概念來實現 value function 的 estimation，以此來讓 board 中各個位置的格子都可以被 tuple 提取到特徵。(這邊對 map 逕行旋轉與鏡像，可以獲得 8 種不同的同構)。

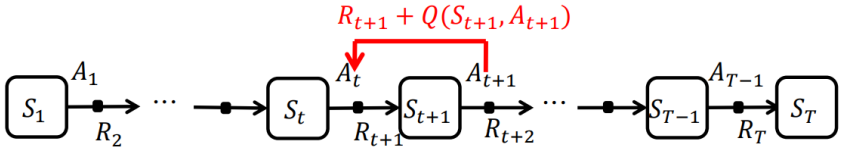
3. Explain the mechanism of TD(0)

TD(0) 可以使用 before state 和 after state 去更新 value function，使用 before state V(s)來更新 value function 的方法稱為 TD(0)的 SARSA，而使用 after state Q(s,a)更新 value function 的方法稱為 TD(0)的 Q-learning。區別在於更新 value function 時，SARSA 會使用實際執行的動作，而 Q-learning 會使用實際執行的最優動作。



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

SARSA 為 state-action-reward-state-action 的簡稱，是一種 on-policy 的算法，在學習過程中使用當前的策略來選擇動作，並使用實際執行的動作來更新值函數，其公式與流程圖如上所示，通常用於有明確動作的問題中，例如棋類遊戲。使用下個 state 的 before state 的值来更新当前 state 和 action 的值，也就是说，我们需要记录 before state、action、reward、after state 和下個 state 的 before state。



$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a} Q(S_{t+1}, a) - Q(S_t, A_t))$$

Q-learning 通過估計動作價值函數 Q 值，即在給定狀態下，採取某個動作所能獲得的長期回報，來選擇最優動作。其為一種 off-policy 算法，在學習過程中使用最優的策略來選擇動作，但是在更新值函數時使用實際執行的最優動作來更新值函數，其公式與流程圖如上所示，適用於存在非明確動作的問題中，例如控制機器人的行動等。

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

想法為根據當下 state 選擇不同 action(上下左右)的 value function(期望值)來判斷走哪個 action，並根據下個 state 的 before state、現在這個 state 的 before 與 reward 來做 weight 的更新，細節如下所示:

```
state select_best_move(const board &b) const
{
    state after[4] = {0, 1, 2, 3}; // up, right, down, left
    state *best = after;
    board next_before_state;
    for (state *move = after; move != after + 4; move++)
    {
        if (move->assign(b))
        {
            // TODO
            // 只有考慮到一種隨機情況的value function
            // next_before_state = move->after_state();
            // next_before_state.popup();

            next_before_state = move->after_state();
            float estimate_value = 0;
            int num = 0;

            for(int i = 0; i < 16; i++, num++){
                if(next_before_state.at(i) == 0){
                    next_before_state.set(i,1);
                    estimate_value += 0.9 * estimate(next_before_state);
                    next_before_state.set(i,2);
                    estimate_value += 0.1 * estimate(next_before_state);
                    next_before_state.set(i,0);
                }
            }
            estimate_value = estimate_value / num;
            move->set_value(move->reward() + estimate_value);
            if (move->value() > best->value())
                best = move;
        }
        else
        {
            // 将当前move的value设置为float类型的负无穷大，为了确保在后续的比较中不会被选择作为最佳的移动选择
            move->set_value(-std::numeric_limits<float>::max());
        }
        debug << "test " << *move;
    }
    return *best;
}
```

上圖為如何選擇 action 的部分，這邊是使用期望值的概念，將當下 board 為 0 的部分進行 2 與 4 的生成(分別為 0.9 與 0.1 機率)，並將其 value function 的值儲存下來。但是，由於每個 state 能夠生成 2 與 4 的位置個數不同，因此來需要除以總共有幾處能生成的數量(類似期望值概念)，最後選擇最高的期望值來做 action 的選擇。

```cpp
void update_episode(std::vector<state> &path, float alpha = 0.1) const
{
    // TODO
    float target = 0;
    // 從最後慢慢往回推
    // state& next_state = path.back();
    for (path.pop_back() /* terminal state */; path.size(); path.pop_back())
    {
        state &state = path.back();
        float error = target - estimate(state.before_state());
        target = state.reward() + update(state.before_state(), alpha * error);
        debug << "update error = " << error << " for" << std::endl
              << state.before_state();
        // next_state = state;
    }
}
```

上圖為 TD-backup diagram 的部分，這邊是從最後一個 state(也就是結束的 state)一個一個 state 慢慢往回更新 weight。這裡的 err 為下個 state 中 before state 的 value function 與當下 state 中的 before state 的差異，這裡的 target 為下個 state 中 before state 的 value function。通過這樣的迭帶，就可以將當下那場遊戲的參數更新完成。

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

上圖為 update 的部分，根據輸入的觀測值，僅更新 network 中特定部分的 weight，以此來改變 agent 的 policy。

```cpp
size_t indexof(const std::vector<int> &patt, const board &b) const
{
    // TODO
    // 该整数的二进制表示方式为将模式字符串中每个字符转换成其在字母表中的数值，
    // 然后将这些数从左到右依次取4个比特位，组成一个长整数
    size_t index = 0;
    for (size_t i = 0; i < patt.size(); i++)
        index |= b.at(patt[i]) << (4 * i);
    return index;
}
```

上圖為特徵編碼的部分，根據給定的 n-tuple 將 board 中特定位置的齊格變成二進位 4bit 的特徵數值，也就是有特徵的地方為 1，沒有的為 0。由於一個 board 至少會有 16 的四次方個特徵，容易使記憶體不足，因此，以此方法來節省儲存空間與加速更新的速度(引用時較快)。