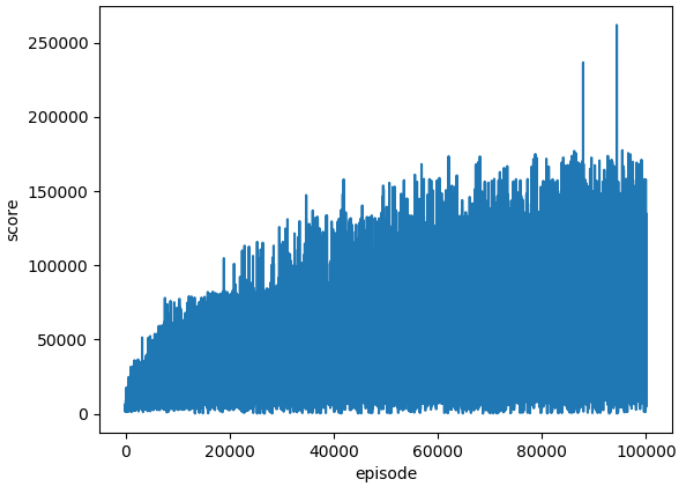
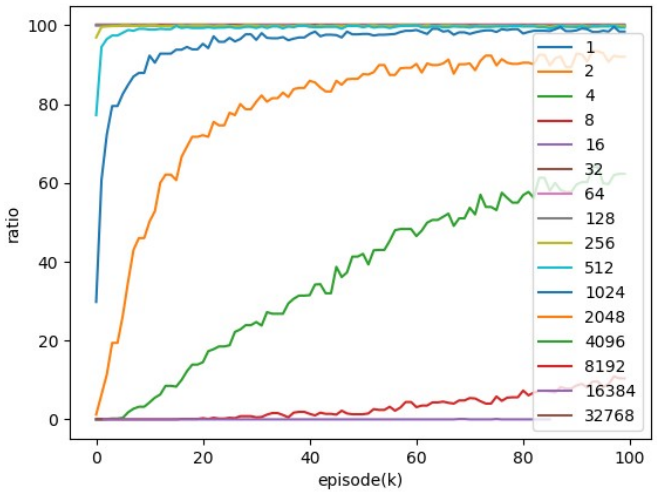
1. A plot shows episode scores





2. Explain the mechanism of TD(0)

TD(0)的公式如右 $V(s_t) = V(s_t) + \alpha(r_{t+1} + V(s_{t+1}) - V(s_t))$

主要想法就是，理想上 V(s_t)會等於 V(s_{t+1})+reward，s_t 和 s_{t+1}分別是當前時刻的盤面和下一個時刻的盤面，而 V 則是一種估測狀態的函式

公式的後面可以看作是 predict error，而 alpha 則是 learning rate，所以整條公式的意義就是希望透過調整 V 對於此刻與下一刻狀態的 predict value，來慢慢將整條 status 的 value 給建出來

3. Describe how to train and use a V(state) network

在 train 的過程跟上面一樣，直接計算前後的差值，並 update value table 裡面的值

TD(0)-state

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

function LEARN EVALUATION(s, a, r, s', s'')
  $V(s) \leftarrow V(s) + \alpha(r + V(s'') - V(s))$

在使用的時候，則是對於每種 action 計算 reward，並判斷 environment 跳出來之後 state 在 value table 裡面的值，尋找最大的那個就執行他

4. Describe how to train and use a V(after-state) network

after-state 在運用時，則是直接找執行 action 後的 reward，加上執行後的 state 在 value table 裡面的值，跟 state network 的差別就是在根據的是環境跳出來前還是跳出來後的 state value

而在 learning 階段則是選一個在 value table 最好的 action，然後計算執行 action 後的盤面與 reward，並更新 value table

TD(0)-afterstate

function EVALUATE(s, a)

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

return $r + V(s')$

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

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

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

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

5. Describe how the code work (the whole code)

最主要的程式在右圖，會有 player 以及 environment 兩個 class，分別代表使用者滑動盤面以及盤面自動跳出方塊兩種動作

而當盤面不能在繼續下去的時候就會跳出中間的迴圈

下面則是關掉各自的 episode，而我們的 update weight 就是做在裡面

```
with player(play_args) as play, rndenv(evil_args) as evil:
    while not stat.is_finished():
        play.open_episode("~:" + evil.name())
        evil.open_episode(play.name() + ":~")
        # type
        ## game => last episode in stat
        ## play => player class (inherit from weight_agent)
        ## evil => rndenv class (inherit from random_agent)
        stat.open_episode(play.name() + ":" + evil.name())
        game = stat.back()
        while True:
            # Play and environment plays in turns
            who = game.take_turns(play, evil)   ## choose who to take action
            move = who.take_action(game.state()) ## input a board as state, return act
            if not game.apply_action(move) or who.check_for_win(game.state()):
                break
        win = game.last_turns(play, evil)
        save_weight = stat.close_episode(win.name())
        total_reward = play.close_episode(stat.back().ep_moves, win.name())
        evil.close_episode(win.name())
        reward_list.append(total_reward)
```

右邊是 player 這個 class，繼承的是 weight_agent，這在下面會再詳細講一下
take_action 實做的是每次決定 action 的情況，會把上下左右四種動作都做一次，然後算 reward 以及後面盤面的 value，最後選最大的那個作為 action

```python
class player(weight_agent):
    """
    dummy player
    select a legal action randomly
    """

    def __init__(self, options = ""):
        super().__init__("name=dummy role=player init=True" + options)
        return

    def take_action(self, state):
        max_value = -1
        max_op = -1
        for op in range(4):
            new_board = board(state)
            reward = new_board.slide(op)
            if reward == -1:
                continue
            else:
                expect = 0
                expect = self.get_value(new_board)
                expect += reward
                if expect > max_value:
                    max_value = expect
                    max_op = op
        if max_op == -1:
            return action()
        else:
            return action.slide(max_op)
```

```python
def init_weights(self):
    self.net += [weight(16777216)]*4

    self.tuples.append([0,1,2,3,4,5])
    self.tuples.append([4,5,6,7,8,9])
    self.tuples.append([0,1,2,4,5,6])
    self.tuples.append([4,5,6,8,9,10])
    self.len_tuples = len(self.tuples)
    return
```

weight_agent 第一個重要的就是 weight 的 initial，這邊我是用 6 tuples 來做

```python
def close_episode(self, ep, flag = ""):
    total_reward = 0
    episode = ep[2:].copy()
    # backward
    ## episode element => (state, action, reward, time usage)
    episode.reverse()
    exact = 0
    for i in range(3, len(episode), 2):
        reward = episode[i-2][2]
        total_reward += reward
        value = self.get_value(episode[i][0])
        error = exact - value
        v = self.alpha * error / 32
        exact = reward + self.update_value(episode[i][0], v)
    return total_reward
```

而 weight 的更新則是做在 close_episode，先把整場遊戲的紀錄最前面兩個挑掉（因為前面兩個動作沒有幫助），然後 reverse 後就開始進入迴圈

因為最後一步並沒有更後面的 state 可以參考了，因此我從倒數第二個 before state 開始更新，value 就是當前盤面在 value table 的值，exact 則是包含了上面的前兩項，error 就是兩者相減了
再來把 error 乘上 learning rate，而除以 32 的原因是我們的 tuple 總共有 32 種可能
最後 update value table，並把 state 最後在 value table 的值回傳，更新 exact 的值給下次使用

```python
def get_index(self, board_state):
    idx_list = [0]*4
    for idx in range(4):
        temp = 0
        for i in self.tuples[idx]:
            temp = temp*16 + board_state[i]
        idx_list[idx] = temp
    return idx_list

def get_value(self, board_state):
    value = 0.0
    for i in range(8):
        new_board = board(board_state)
        if (i >= 4):
            new_board.transpose()
        new_board.rotate(i)
        idx_list = self.get_index(new_board)
        for weight_idx, idx in enumerate(idx_list):
            value += self.net[weight_idx][idx]
    return value
```

上面是整個 update weight 的架構，實做面在左圖三個 function

首先是 get_value，取得 state 的 value
最外層迴圈是考慮了八種 tuple 可能的轉法（四種方向轉，以及鏡像之後四種方向，共八種）
每次進去就複製一個新的盤面，然後對他做相對應的動作，等同於讓 tuple 做旋轉取值
然後就進到 get_index，取得每個 tuple 對於這個盤面的值
最後就把 self.net 裡面的值取出來相加，回傳就是整個盤面的 value 了

```python
def update_value(self, board_state, value):
    total = 0.0
    for i in range(8):
        new_board = board(board_state)
        if (i >= 4):
            new_board.transpose()
        new_board.rotate(i)
        idx_list = self.get_index(new_board)
        for weight_idx, idx in enumerate(idx_list):
            self.net[weight_idx][idx] += value
            total += self.net[weight_idx][idx]
    return total
```

再來是 update value 的部份
跟 get_value 很像，差別只在最後每個 tuple 的 weight 都要加上剛算出的 alpha*error/32
total 則是為了回傳更新後的盤面值，方便更新下一個 episode element

右邊則是 environment 在處理隨機跳出 tile 的動作

在 take action 裡面，一開始是先找哪邊有空的位置，然後就是隨機在這些位置裡面選一個，0.9 的機率放 2-tile，0.1 機率放 4-tile

到目前為止就是整個程式大概的運作，剩下的就是盤面以及 episode，就不再贅述了

```python
class rndenv(random_agent):
    """
    random environment
    add a new random tile to an empty cell
    2-tile: 90%
    4-tile: 10%
    """

    def __init__(self, options = ""):
        super().__init__("name=random role=environment " + options)
        return

    def take_action(self, state):
        empty = [pos for pos, tile in enumerate(state.state) if not tile]
        if empty:
            pos = self.choice(empty)
            tile = random.choice([1] * 9 + [2])
            return action.place(pos, tile)
        else:
            return action()
```

6. More you want to say

這個作業因為有 code 了，所以一開始就在 trace code，python 版本的 code 還滿好讀的，相比下來 c++因為都寫在一起，所以就選擇 python 作為這次作業的語法了

在開始寫的時候，第一個問題是一開始的 tuple 沒有做旋轉和鏡射的八種狀況，所以雖然 accuracy 起得來，不過成效並不好，後來才想到應該要做其他種可能

但第二個問題就是不知道 tuple 在旋轉後，到底是要 32 種 weight 還是 4 種 weight，最後是看到原本 code 就有寫好取得盤面的範例 function，才改成四種 weight 的版本，也才 train 起來這個作業

7. Strength

1024-tile => 99.5%