

俄罗斯方块

俄罗斯方块

[任务描述](#)
[代码填空理解](#)
[训练与测试](#)
[结果](#)
[结论与反思](#)
[附录](#)

全文共3000词，阅读需要十分钟。完成人@陈屹峰20373870。

任务描述

在 `missionXML` 中定义了环境：画出俄罗斯方块的基本游戏背景

主要任务：

Agent只有在俄罗斯方块下落时才能操纵它，掉落的方块可以旋转、水平移动或掉落到游戏区域的底部。在整个游戏过程中，俄罗斯方块从游戏区域的顶部落到底部，当俄罗斯方块在游戏区形成一排实心方块时，该行消失。Agent需要学习如何控制下落的俄罗斯方块，使得尽可能地在一局游戏中**消去尽可能多的行数、达到尽可能多的下落块数**。

相关定义：

```
# 游戏区尺寸列宽为5，高度为12
cols = 5
rows = 12
```

```
# 奖励字典 有增加高度，清除行数，产生空洞和最高高度四种环境反馈
rewards_map = {'inc_height': -20, 'clear_line': 30, 'holes': -20, 'top_height': -100}
```

每局胜负判断：

```
check_collision() # 假如board和该块碰撞，则游戏结束
```

代码填空理解

```
class TetrisGame:
    def move(self, delta_x): # 已经得到偏移量delta_x，对于块的移动
        if not self.gameover:
            new_x = self.piece_x + delta_x
            # 确保不会把块移出边界
            if new_x < 0:
                new_x = 0
            if new_x > cols - len(self.piece[0]):
                new_x = cols - len(self.piece[0])
            # check_collision函数检查游戏是否结束
```

```

        if not check_collision(self.board,
                                self.piece,
                                (new_x, self.piece_y)):
            self.piece_x = new_x
    def draw_piece(self):
        for cy, row in enumerate(self.piece): # 枚举每行
            for cx, col in enumerate(row): # 枚举每列
                if col != 0:
                    self.agent_host.sendCommand("chat /setblock " + str(0 +
self.piece_x + cx) + " "
                                                    + str(80 - self.piece_y - cy) + " 3
wool " + str(col))

    def draw_piece2(self, piece, x=0):
        for cy, row in enumerate(piece):
            for cx, col in enumerate(row):
                if col != 0:
                    self.agent_host.sendCommand("chat /setblock " + str(0 + cx +
x) + " "
                                                    + str(80 - cy) + " 3 wool " +
str(col))
                elif col == 0:
                    self.agent_host.sendCommand("chat /setblock " + str(0 + cx +
x) + " "
                                                    + str(80 - cy) + " 3 air")

```

```

class TetraAI:
    def run(self, agent_host):
        states, actions, rewards = deque(), deque(), deque() # 建立双向列表
        curr_reward = 0 # 当前回报
        done_update = False # 结束整个训练/测试的变量
        game_overs = 0
        self.loadQtable() # 可以装入以前训练的Q表
        while not done_update:
            init_state = self.get_curr_state() # 当前状态s
            possible_actions = self.get_possible_actions() # 可行的动作集A
            next_action = self.choose_action(init_state, possible_actions) # max
Q(s,a), a in A
            states.append(init_state)
            actions.append(self.normalize(self.pred_insta_drop2(next_action))) #
将next_action映射到下一状态next_state, 并存在actions列表中
            rewards.append(0)

            T = sys.maxsize # 这里意在使得训练持续进行。测试时可以将其改为0
            # T = 1000
            # for t in range(T)
            for t in range(sys.maxsize):
                time.sleep(0.1)
                if t < T:
                    curr_reward = self.act(next_action) # 当前奖励r 并将动作后的情形
绘制到Minecraft中
                    rewards.append(curr_reward)

```

```

#####
if self.game.gameover == True: # 如果结束了
    game_overs += 1
    self.gamesplayed += 1 # 总游戏次数
    self.listGameLv1.append(self.game.level)
    self.listClears.append(self.game.line_clears)
    print("这把agent到了第:", self.game.level, "关")
    print("这把agent消去了", self.game.line_clears, "行")
    self.game.start_game() # 重开一局

    if game_overs == 10: # 每十局游戏打印目前战况
        print("到目前为止,agent训练局数: ", self.gamesplayed,
              "平均关数为", numpy.mean(self.listGameLv1),
              "平均消去行数为", numpy.mean(self.listClears))
        self.saveQtable() # 保存Q表
        game_overs = 0

    curr_state = self.get_curr_state()
    states.append(curr_state)
    possible_actions = self.get_possible_actions()
    next_action = self.choose_action(curr_state,
possible_actions)

    actions.append(self.normalize(self.pred_insta_drop2(next_action)))
#####

    tau = t - self.n + 1
    if tau >= 0:
        self.update_q_table(tau, states, actions, rewards, T) # 更新Q
表

    if tau == T - 1:
        while len(states) > 1:
            tau = tau + 1
            self.update_q_table(tau, states, actions, rewards, T)
        done_update = True # 如果达到了训练幕数则结束训练
        break

    if t%5000 == 0:
        self.saveQtable()
        print("-----Saving Qtable-----")
        time.sleep(0.1)

def update_q_table(self, tau, S, A, R, T):
    curr_s, curr_a, curr_r = magic(S.popleft()), A.popleft(),
R.popleft() # 从状态集, 动作集和奖励集中取出当前值
#####
    next_s = magic(curr_a) #
actions.append(self.normalize(self.pred_insta_drop2(next_action)))此处取出的即是下一个状态

    G = sum([self.gamma ** i * R[i] for i in range(len(S))])
    if tau + self.n < T:
        G += self.gamma ** self.n * self.q_table[magic(S[-1])]
[magic(A[-1])]

```

```

        old_q = self.q_table[curr_s][next_s] #
        self.q_table[curr_s][next_s] = old_q + self.alpha * (G - old_q)
        #####
def choose_action(self, state, possible_actions):
    curr_state = magic(state)
    #####
    # 如果没有则创建一个Q表项
    if curr_state not in self.q_table:
        self.q_table[curr_state] = {}
    for action in possible_actions:
        next_state = magic(self.normalize(self.pred_insta_drop2(action)))
        if next_state not in self.q_table[curr_state]:
            self.q_table[curr_state][next_state] = 0

    rnd = random.random()
    if rnd < self.epsilon: # ε-greedy
        a = random.randint(0, len(possible_actions) - 1)
        best_action = possible_actions[a]
    else:
        best_actions = [possible_actions[0]]
        best_next_state =
magic(self.normalize(self.pred_insta_drop2(best_actions[0])))
        qvals = self.q_table[curr_state]
        for action in possible_actions:
            next_state =
magic(self.normalize(self.pred_insta_drop2(action)))
            if qvals[next_state] > qvals[best_next_state]: # 从Q表中找到最好状态
                best_actions = [action]
                best_next_state = next_state
            elif qvals[next_state] == qvals[best_next_state]:
                best_actions.append(action)
        a = random.randint(0, len(best_actions) - 1) # 在最优动作中随便选一个
        best_action = best_actions[a]
    #####
    return best_action

def score(self, board):
    current_r = 0
    complete_lines = 0
    highest = 0
    holes = 0

    #####
    new_board = board[-2::-1]

    # 检查是否有完整的一行
    for row in new_board:
        if 0 not in row:
            complete_lines += 1
    current_r += complete_lines * rewards_map['clear_line']

    # 检查使得已有的块总增高了多少
    heights = zeros((len(new_board[0]),), dtype=numpy.int)
    for i, row in enumerate(new_board):
        for j, col in enumerate(row):

```

```

        if col != 0:
            heights[j] = i + 1
        current_r += sum(heights) * rewards_map['inc_height']

# 检查最高的点增高了多少
for i, row in enumerate(new_board):
    if all(j == 0 for j in row):
        highest = i
    current_r += (highest - 2) * rewards_map['top_height']

# 检查空洞
for i, row in enumerate(new_board):
    if 0 in row:
        row_indexes = [i for i, j in enumerate(row) if j == 0]
        for index in row_indexes:
            if index == 0:
                if new_board[i][1]:
                    holes += 1
            elif index == len(row) - 1:
                if new_board[i][index-1]:
                    holes += 1
            elif new_board[i][index-1] and new_board[i][index+1]:
                holes += 1
        if all(j == 0 for j in row):
            break

    current_r += holes * rewards_map['holes']
#####

return current_r

```

训练与测试

采用Q学习方法，建立动作价值函数表，进行重复训练，并在每10次训练之后将q表储存在 `.save` 文件中。测试时可以装入Q表，从而将训练和测试相隔离。

关于奖励，这里考虑四种情况：

如果能够完整清除一行，则奖励30，如果使得整体的高度增加，则每增加1格奖励-20，如果使得存在空洞，则每个空洞-20，最高高度奖励-100

关于状态和动作的存储方式，状态存储采取简化方法，仅关心上面两行的情况，即有方块为一，无方块为零

0	1	0	1	1	1	0	1	1	0
---	---	---	---	---	---	---	---	---	---

动作的存储方式，由于存在不同种类的方块以及存在同样的操作给出相同的结果的动作，这里不能够将动作简单理解为

“位移和旋转”的二元组，因为对于相同状态，由于会出现不同的方块，对于不同的方块应该是有着不同的动作。同时考虑到我们上述的情况状态存储的情况，我们将动作定义为考虑到当前状态、当前下落方块以及施加“位移和旋转”操作之后的，所得的下一状态，将这个信息存储到Q表中。将操作映射为动作并存储的函数是 `self.normalize(self.pred_insta_drop2(next_action))`。

结果

参见视频，在训练了40000左右局的时候，agent平均每局下落16块俄罗斯方块，平均每局消除层数为0.9，训练时间约为一小时。

对比未训练的agent，采用随机动作，平均每局下落10块俄罗斯方块，平均每局消除层数为0.2。

这样的结果能够明显说明agent学习到了俄罗斯方块的技术。

结论与反思

基于上两层状态和给定奖励的Q-learning学习到了俄罗斯方块的技巧。


实践层面

对于一个问题，采用良好的数据结构和算法流程是解决他的先决条件。本次实验历时很久，主要原因在于没有很好的理解整个实验各个函数的接口。在今后的代码写作过程中应该尽量捋清楚各个函数的调用关系再开始写作。

模型层面

虽然agent能够比未训练的agent得到较好的技巧，但是由于训练收敛时间过长，这里需要思考怎样能够加速训练过程，以及怎样使得agent在训练的过程中能够更好地利用人类的先验知识，例如俄罗斯方块中的一些固定的玩法套路。

这里给出了对于纯粹的QLearning方法的改进。我将一些情况进行总结，给出一些固定的游戏规则：

类型	规则
	选择较深的位置插空，若无较深的空，则随机插入最上层的空
	优先插入竖着两行的空，若没有则插入横向的两格的空，若没有，则旋转90°并随机插入最上层的空
	查看最上层是否有 $\begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$ 或者 $\begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$ ，如果有则将方块调整至合适的位置，若无，则随机处理
	同上
	选择横向的两格，若没有则随机处理

```
def apriori(self):
    type = get_piece_type(self.game.piece)
    state = self.get_curr_state()
    if type == 1:
        for j in range(self.game.rlim):
            if state[0][j] + state[1][j] == 0:
                return [j, 0]
        temp = []
        for j in range(self.game.rlim):
            if state[0][j] == 0:
                temp.append(j)
        return [temp[random.randint(0, len(temp) - 1)], 0]
```

```

if type == 2:
    for j in range(self.game.rlim):
        if state[0][j] + state[1][j] == 0:
            return [j, 1]
    for j in range(self.game.rlim - 1):
        if state[0][j] + state[0][j+1] == 0:
            return [j, 0]
    for j in range(self.game.rlim):
        if state[0][j] == 0:
            return [j, 1]

if type == 3:
    for i in range(self.game.rlim - 1):
        if state[0][i] == 0 and state[0][i+1] == 0 and state[1][i] == 1
and state[1][i+1] == 0:
            if self.game.piece[0][1] == 0:
                return [i, 0]
            else:
                return [i, 1]
        if state[0][i] == 0 and state[0][i + 1] == 0 and state[1][i] ==
0 and state[1][i + 1] == 1:
            if self.game.piece[0][1] != 0:
                return [i, 0]
            else:
                return [i, 1]
    return [random.randint(0, self.game.rlim - 1) - self.game.piece_x,
random.randint(0, 1)]

if type == 4:
    for i in range(self.game.rlim - 1):
        if state[0][i] == 0 and state[0][i+1] == 0 and state[1][i] == 1
and state[1][i+1] == 0:
            for k, row in enumerate(self.game.piece):
                for j, col in enumerate(row):
                    if col == 0:
                        if k == 0 and j == 0:
                            return [i, 3]
                        elif k == 0 and j == 1:
                            return [i, 2]
                        elif k == 1 and j == 0:
                            return [i, 0]
                        else:
                            return [ix, 1]

                    elif state[0][i] == 0 and state[0][i + 1] == 0 and state[1][i]
== 0 and state[1][i + 1] == 1:
                        for k, row in enumerate(self.game.piece):
                            for j, col in enumerate(row):
                                if col == 0:
                                    if k == 0 and j == 0:
                                        return [i, 2]
                                    elif k == 0 and j == 1:
                                        return [i, 1]

```

```

        elif k == 1 and j == 0:
            return [i, 3]
        else:
            return [i, 0]

    else:
        return [random.randint(0, self.game.rlim - 1),
random.randint(0, 1)]

    if type == 5:
        for j in range(self.game.rlim - 1):
            if state[0][j] + state[0][j+1] == 0:
                return [j, 0]
        return [random.randint(0, self.game.rlim - 1), 0]

```

在这样的训练条件之下，平均消去块数为1.4块，明显高于AI，但也是明显低于一个俄罗斯方块的新手。这里也引起我们的思考，就是完全没有先验知识的Q学习agent是否能够再一定的时间内超过具有先验知识的agent，也即在游戏空间很大的时候选择较为简单的强化学习算法是很难与人类的游戏能力相比的。如果要进一步提高agent的能力，则需要使用DQN+CNN等基于神经网络的方法对动作价值函数进行估计。

附录

本实验压缩包中附有代码两套、训练文件两套、展示PPT一个、实验报告一篇。

本实验附有实验代码两套：

`tetrisAIQL.py`, `tetris_game.py` 和 `tetrisAIQL2.py`, `tetris_game2.py`

分别是Q-learning和完全先验算法，在测试时分别在跳板机中直接运行 `python3 tetrisAIQL*.py` 即可。如果想要重新训练agent，请注释掉 `loadQtable()` 函数，即可覆盖原来的训练记录进行训练。

`maltrisQLTable.save` 和 `maltriQLGraphData.save` 分别是训练存储的Q表。