

华南师范大学

《人工智能导论》课程项目

课 程 项 目 报 告

项 目 题 目：基于 Q-Learning 实现 Doodle Jump 自动跳跃

所 在 学 院：计算机学院

项 目 组 长：宋轶德

小 组 成 员：陈远镇、唐坤生、罗添、彭学典

开 题 时 间：2020 年 4 月 1 日

目录

一、引言	1
二、国内外研究现状	1
1. 目前人工智能算法分类	1
2. Q-Learning 模型研究	1
(1) Q-Learning 模型介绍	1
(2) Q-Learning 模型核心步骤	2
三、模型和算法	3
1. 模型概要	3
(1) 主要构成	3
(2) 模型流程	4
(3) 模型算法伪代码	4
2. 环境介绍	4
(1) 组成元素	4
(2) 游戏操作	5
(3) 应用 Qlearning 后的踏板属性	6
3. Qlearning 具体引用	6
(1) 状态定义	6
(2) 预测函数 predict	7
(3) 奖赏反馈函数	8
(4) 获取状态函数	8
(5) 选择函数	10
(6) 移动方向函数	12
(7) 保存训练数据	13
4. 数据可视化	13
5. 版本控制	14
6. 人机对战界面效果	14
四、实验结果分析	16
五、结论	16
六、参考文献	17

一、引言

游戏与人工智能作为近几年均获得飞速蓬勃发展的行业，两者的融合却一直没有获得明显的进展。“愚钝”的游戏人机总是遭到多数游戏玩家的唾弃。优秀的人工智能“玩家”不仅能良性地引导新手玩家获得游戏的乐趣，还能协助现有玩家探索游戏更多的可玩性。基于上述需求，本小组以游戏《涂鸦跳跃(Doodle Jump)》为例，建立一个人机模型，采用了强化学习中的 Q-Learning 算法进行模型训练，并对训练好的模型进行一定的观察。最后，模型的表现也验证了所提想法的有效性。

二、国内外研究现状

1. 目前人工智能算法分类

目前人工智能算法有三大种类：分别是有监督学习，无监督学习，以及强化学习。与有监督学习相比，强化学习通过自身不断的犯错和奖赏的获取来获得优化和提升。也就意味着，强化学习不用提供庞大的数据集。强化学习中也分为有模型学习和无模型学习。有模型算法的优点是，可以提前考虑并进行规划，训练效率比较高。缺点是需要对环境有提前的认知，如果模型跟现实世界不一一致，会导致在实际使用场景下会表现的不好。无模型算的优点是，容易实现，而且在真实场景中更好调整到好的状态。但训练效率跟有模型的算法那相比会有短板。目前，人们对无模型算法更为重视。因为对真实环境的适应能力真的很重要。这次题目，我们所采用的是强化学习这一大类中的无模型学习中的 Q-Learning 模型。虽然，我们这次题目并不用考虑对现实环境的适应，但其简单性以及丰富的资源正是我们想要的。

2. Q-Learning 模型研究

(1) Q-Learning 模型介绍

Q-learning 模型运行的本质是不断的尝试并根据环境的反馈和自己的估计进行比较，修正自己的估计值和做事方法。经过很多次的修正后，Q-learning 便逐渐有一套正确的做事法则。下面给出 Q-learning 的定义和概览：

Q-Learning 算法是强化学习算法中的一种，其中的 Q 值是在某一状态下采取某种动作能够获得收益的期望，同时 Q 值会根据环境的状态反馈相应的操作信息。在 Q Learning 算法中，“状态 State”与“行为 Action”是两个重要的单位，需要完成的目标为“状态”，完成的路径为“行为”，两者构建成一张 Q-table 来储存 Q 值，依照 Q 值来选取能够获得最大收益的操作[1]

(2) Q-Learning 模型核心步骤

Q-Learning 核心是维护一个 Q-Table，可以理解为 Q-learning 的做事法则，其中 S 是状态集，A 是动作集。

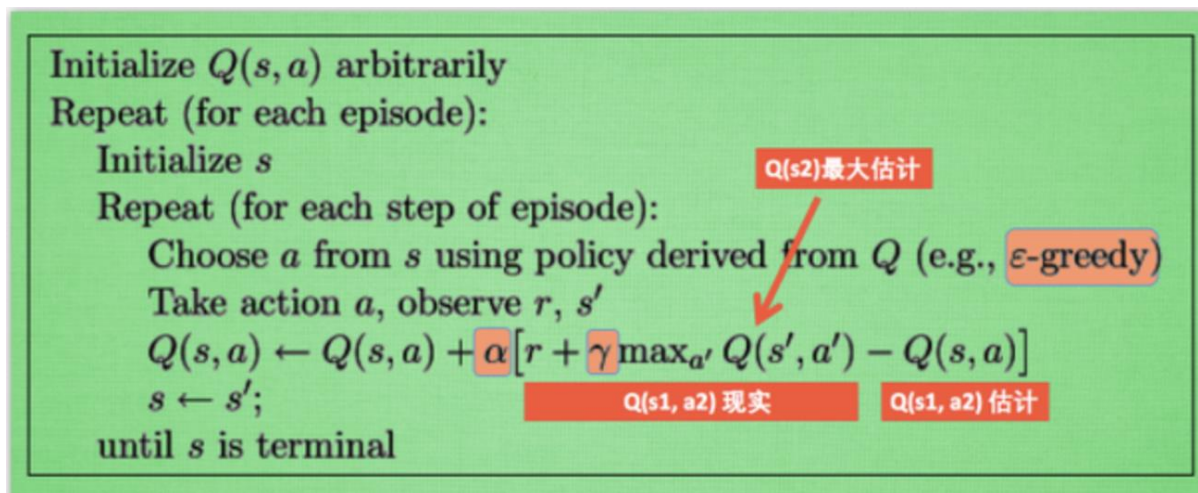


图 1 Q-Learning 的核心算法伪代码

符号	说明
s	当前状态
s'	经过行为后达到的状态
a	行为
r	这次行为所获得奖励
$\max Q(s', a')$	在 Q 表中, 找到执行行为 a 后, 达到状态 s' , 基于该状态所能获得的最大奖励
$Q(s, a)$	估计位于上一个状态, 做出决策能获取的奖励
e	随机选择当前决策的概率
$1 - e$	基于当前估计选择最优状况的概率
α	学习效率, 决定当前误差的学习程度 < 1
γ	衰减值。对未来奖励重视程度。 γ 值越大, 对未来奖励越重视

表 1 公式符号解释

与神经网络类似，背后的准则是：学习率 \times (真实值 - 预测值)。Q-learning 模型每一次做出决策时，总会根据之前所做过的决策，判断当前状态下，哪一个行为能获取的利益越大，甚至可估算是哪个行为对未来总的收益越大，然后做出行为决策。当然，Q-learning 也有一定概率会随机做出决策。做出决策后，会得到真实的回报值，得到该回报值后再与估计值对比，修正估计值。

三、模型和算法

1. 模型概要

Q-learning 是一种无模型强化学习算法，用于学习特定状态下动作的价值。它不需要环境模型（因此是“无模型”），它可以处理随机转换和奖励的问题，而无需进行调整。

对于任何有限马尔可夫决策过程(FMDP)，从当前状态开始，Q-learning 发现了一种最优策略，即从当前状态开始，在所有连续步骤中最大化总奖励的期望值[2]。Q-learning 可以为任何给定的 FMDP 确定最佳动作选择策略，给定无限探索时间和部分随机策略[2]。“Q”指的是算法计算的函数——在给定状态下采取的行动的预期回报。

早在 2016 年，Elliot Shohet 曾使用 JavaScript 完成 Doodle Jump 和 Q-learning 的开发，并将两者结合在一起。实现一个玩 Doodle Jump 跳跃游戏的超强人机。参考他们的经验，本文采用的 Q-learning 模型介绍如下。

（1）主要构成

构成部分	作用
记录状态表 QTable	记录环境中的每一个状态的分数，即 Q 值
预测函数 predict	输入环境状态，输出基于当前 QTable 的最优解
奖赏函数 reward	输入环境反馈，修正 QTable 的值

表 2 模型主要构成部分介绍

(2) 模型流程

Qlearning 是强化学习算法中 value-based 的算法。智能体每一次会基于当前环境情况，调用预测函数，从 QTable 中挑选出最优的状态执行。执行完毕后，环境会给智能体一个反馈，即奖赏。智能体会根据奖赏修正 QTable，以便下一次做出更好的选择。

(3) 模型算法伪代码

```
初始化 QTable
While(每一轮环境)
    初始化当前状态 s
    While(到达新的状态)
        从 Qtable 中选出最优的行为 a
        执行行为 a，获取状态 s'，获取回馈 r
         $Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ 
         $s = s'$ 
    状态不可往下走
该轮结束
结束学习

参数说明：
 $\gamma$ : 衰减值 对未来奖励的重视程度，
 $\alpha$ : 学习率 对这次奖赏受影响的程度。
```

这是 Qlearning 的算法框架，很多时候需要根据具体的环境而进行修改。

2. 环境介绍

(1) 组成元素

① 玩家

可以往左，往右走。碰到可以跳跃的平台会往上跳，到达最高顶点后，会往下移动。



图 2 玩家 player

② 踏板

类型	功能	图示
普通踏板	可以无限次的踩踏，可以跳跃	
一次性踏板	只可以踩踏一次，踩踏后消失，可跳跃	
陷阱踏板	踩踏一次会消失，且不可跳跃	

表 3 踏板类型介绍

③ 弹簧

可踩踏，可跳跃，且跳跃提升的高度与普通踏板要更高。

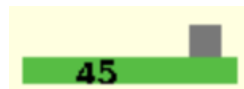


图 3 弹簧图示

④ 分数展示

分数会展示在游戏界面的左上角，并会随着玩家跳的高度计算并动态刷新。玩家跳的越高，分数越高

2 m

图 4 分数展示示意

(2) 游戏操作

根据键盘左右键位，操纵玩家左右移动。当玩家跌落到窗口的最低点时，则判断玩

家死亡，结束该轮游戏并显示分数。

(3) 应用 Qlearning 后的踏板属性

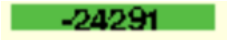


类型	功能	图示
踏板分数	每一个踏板都会对应有一个 Q 值	
红色踏板	当前智能体做出的选择	
黑色踏板	智能体上一次踏上的踏板	

表 4 应用 Q-learning 后的踏板属性介绍

3. Qlearning 具体引用

(1) 状态定义

我们选取游戏中三个比较重要的因素：

- ① 平台种类
- ② 平台离玩家 y 距离
- ③ 平台离玩家的 x 距离

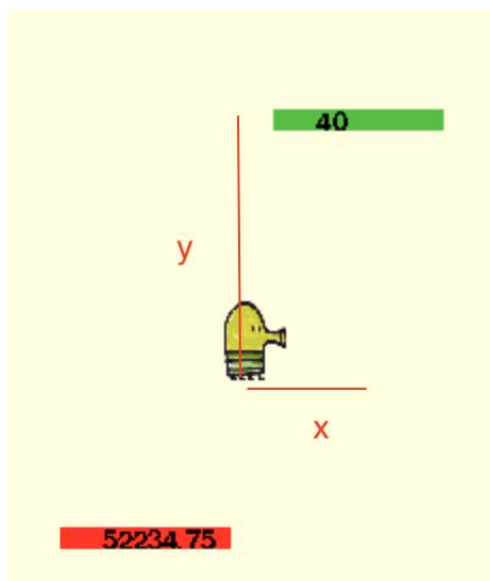


图 5 重要因素演示

代码中，我们使用 python 的 dict 类型存储 Qtable。初始化为空 dict。若发现新的状态，则会将该状态分数设置为范围于[0, 100]的随机数。

(2) 预测函数 predict

获取平台的状态，并以此为参数输入到 predict 函数，predict 会从 Qtable 中获取分数返回并记录住当前的状态。

```
def predict(self, state):
    self.last_state = state
    # 平台类型
    i = state[0]
    # 平台的 y 距离
    j = state[1]
    # 平台的 x 距离
    k = state[2]

    # 处理已经见过的平台
    if i in self.action:
        # 处理该平台下的 y 距离
        if j in self.action[i]:
            # 处理该该平台下 y 距离和 x 距离
            if k in self.action[i][j]:
                return self.action[i][j][k]
            else:
                # 新的 x 距离 添加随机值
                self.action[i][j][k] = 0
                self.action[i][j][k] += ra.randint(0, 100)
                self.explored += 1
                return self.action[i][j][k]
        # 新的 y 距离
        else:
            self.action[i][j] = {}
            self.action[i][j][k] = 0
            self.action[i][j][k] += ra.randint(0, 100)
            self.explored += 1
            return self.action[i][j][k]
    # 新的平台类型
    else:
        self.action[i] = {}
        self.action[i][j] = {}
```

```
self.action[i][j][k] = 0
self.action[i][j][k] += ra.randint(0, 100)
self.explored += 1
return self.action[i][j][k]
```

(3) 奖赏反馈函数

输入分数 amount，修正上一个状态的分数。上一个状态的分数 += 学习率 * amount。同时，达到某些条件后，也会修改上一个状态的分数。防止智能体一直卡在同一个状态。

```
# 奖赏反馈
def reward(self, amount):
    # positive 为 1, 随机执行
    positive = 0

    i = self.last_state[0]
    j = self.last_state[1]
    k = self.last_state[2]

    if self.action[i][j][k] > 0:
        positive = 1

    self.action[i][j][k] += self.learning_rate * amount

    # 随机执行
    if self.action[i][j][k] == 0 and positive == 1:
        self.action[i][j][k] -= 1
```

(4) 获取状态函数

输入平台列表。遍历每一个平台 p。

记录[p 的种类, 取整 ((p 的 y 轴 - 玩家 y 轴) / yDivision) , 绝对值和取整 (p 的 x 轴 - 玩家的 x 轴) / xDivision]

```
# 获取状态
def get_states(platforms, player):
    state = {}
    # 设置 y 轴距离, 简化状态空间
```

```
yDivision = 1
# 设置 x 轴距离, 简化状态空间
xDivision = 5
for platform in platforms:
    state[platform] = [platform.kind(), round((platform.posY() -
player.posY()) / yDivision), abs(round((platform.posX() - player.posX()) /
xDivision))]
return state
```

该游戏的界面宽为 640 像素, 长为 700 像素, 平台种类一共有三种。根据状态空间定义, 可得状态空间大小为 $640 * 700 * 3 = 1\,344\,000$ 。共有 1344000 种状态, 需要智能体去探索, 而且每种状态还需访问多次才能修正为较为正确的值。在训练过程中, 我们将游戏速度加快为相对于现实时间的 50000 倍, 即游戏中 50000s 等于现实 1s。采用 1.4 GHz Intel Core i5 处理器, 运行达两小时, 游戏进行 25 万轮后, 游戏状态才探索到 5856703 轮。所以我们对状态空间进行适当的压缩。y 距离不压缩, x 距离会除 5 后取整数部分。如: 平台离玩家的 x 距离为 [160, 164] 都会视为 160。不可过分压缩空间, 不然会出现如下情况。

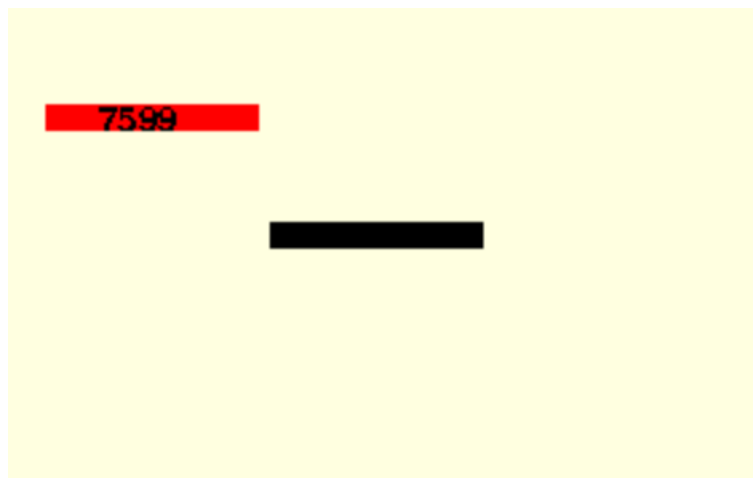


图 6 异常情况示意图

黑平台和红平台均视为同一种状态, 但有时候之智能体在某些状态下可以到达黑平台, 而到达不了红平台。但红黑平台的预测分数是一致, 这时就会出现问题。预测分数很高, 但是智能体无法到达, 导致智能体死亡。

(5) 选择函数

decide 函数整合 predict 函数和 reward 函数。

输入：平台列表，玩家，当前游戏的分数，玩家上一次抵达的平台

① 判断玩家是否死亡：

已经死亡，给与惩罚，惩罚分数为 $-500 * (1 + \text{当前分数} / 2000)$ 。

② 若到达的平台不是预测的平台：

若目标平台的高度要低于当前到达平台的高度：给予惩罚，惩罚分数为 20 分。

当前到达的平台的高度比目标平台要高，说明选取的目标平台并不是很好的选择。

若不是：也给予惩罚，惩罚分数为 10 分。这样做是为防止智能体选取根本无法到达的平台，而一直原地跳跃。

很多时候，当玩家往目标平台移动时，途中可能会碰到其他踏板而跳跃。所以需要根据玩家真正碰到的平台进行反馈。

③ 对当前到达的平台进行反馈

获取当前到达的平台的状态。值得注意的是，不是当前玩家的状态和当前碰撞平台的状态，而是上一次做出预测时，当前碰撞平台的和那时玩家的状态。分数 $r = \text{当前分数} - \text{上一次分数} - 20$

调用 reward 函数，传入 r。

④ 更新平台状态

传入平台列表并输入到 get_state 函数，获取状态并记录。若平台离玩家太高，是不予考虑的。

⑤ 挑选目标平台

利用 predict 函数获取当前所有平台的分数，并挑选出分数最高的平台为目标平台。并将 predict 的上一个状态记录为目标平台与当前玩家的状态。

⑥ 特殊处理

由于弹簧并没有考虑到范围之内，且弹簧给与的反馈力度过大，会对 Qtable 有较大的干扰。所以弹簧只会预测而不会进行反馈。

更新目标平台的时间。我们并不是每时每刻都进行目标平台的预测，当玩家碰到一个踏板后，才会执行预测函数。如果是每时每刻都执行预测，智能体会因为

目标平台的切换速度过快而待在原地。

```
# 遍历当前的所有平台，然后预测每一个平台的分数
# previous_collision 需要从外面更新，因为 target_platform 不一定是当前平台
def decide(platforms, player, score, real_platform, counter=1, isBounce=False):
    global previous_score
    global states
    global target_platform
    global last_platform
    if isBounce is False:
        if player.dead:
            if target_platform in states:
                brain.predict(states[target_platform])
                brain.reward(-500)
            # if real_platform in states:
            #     brain.predict(states[real_platform])
            #     brain.reward(-500)

            previous_score = 0
            target_platform = None
            return

        if target_platform is not None and real_platform is not None:
            # 到达的平台不是目标平台
            if real_platform != target_platform and real_platform in states and
target_platform in states:
                if target_platform.posY() > real_platform.posY():
                    brain.reward(-200)
                else:
                    brain.reward(-100)

            if real_platform is not None and real_platform in states:
                brain.predict(states[real_platform])
                # 防止初始分数带来过强的干扰
                if previous_score == 0 and score != 0:
                    previous_score = score - 20

            r = score - previous_score - 20
            # 追加限制
            if r > 200:
                r = 200
            # print(r)
            brain.reward(r)
```

```
        previous_score = score
        last_platform = real_platform
        # ###
        # if real_platform.kind() == 2:
        #     brain.reward(-10000)

    else:
        previous_score = score

    states = get_states(platforms, player)
    maxReward = -float('inf')

    # 遍历平台 并从总挑选预测分数最高的平台
    for zz in range(0, len(platforms)):
        if player.posY() - 500 <= platforms[zz].posY() <= player.posY() + 300:
            platforms[zz].predictScore = brain.predict(states[platforms[zz]])
            if maxReward < platforms[zz].predictScore:
                maxReward = platforms[zz].predictScore
                target_platform = platforms[zz]

    # 调用预测函数, 将当前平台更新为上一个平台
    if target_platform is not None:
        brain.predict(states[target_platform])
```

(6) 移动方向函数

输入：玩家

根据玩家当前 x 轴与目标平台的 x 轴做出方向移动。若平台位于玩家的左侧，则往左移动，若位于右侧，则往右移动。

若玩家已经在平台范围之内，则会停止移动。该功能的实现是嵌入游戏主循环中实现。

```
# 选择方向移动
# 挑选出最适合的平台, 比较 player 和该平台的 x 距离, 做出移动
# none 为不移动
def direction(player):
    dire = "none"
    if target_platform is None:
        return dire, target_platform
```

```
pX = target_platform.posX()
# 防止 player 跳过平台
if pX <= player.posX():
    dire = "left"
elif pX > player.posX():
    dire = "right"

return dire, target_platform
```

（7）保存训练数据

该模型最为重要的数据是 QTable 中各状态的分数，而该分数数据的存储，我们采用的是 pickle 模块，python 序列化模块，将数据保存当当前目录中的 Qdict.txt，并同时生成 json 格式的数据，便于我们查看。QTable.txt 则是记录智能体的探索状态。若将三个文件全部删除则会重新训练。

4. 数据可视化

为了知道训练是否有其作用，其效果直接体现在游戏分数上，我们希望能够可视化分数的变化过程，即每一局游戏获得的分数是否在增长。不同于 JS 版本的实现，python 的游戏运行在 pygame 里，分数展示的部分不容易直接获得这个进程里面的游戏分数。所以我修改了游戏逻辑：在每一次游戏结束后，执行 SQL 语句，将该局游戏分数加入到数据库里。

分数展示的进程是独立于游戏运行的进程的，我选择使用了 Dash (plotly) 这个框架。plotly 是一个图表库，然而这是静态的：将数据输入，并生成静态的 html/png，不便于展示动态变化的过程。所以我了解到了同一个团队出的 Dash。Dash 是基于 Flask, plotly.js, react.js 的 python 框架，对于我们这个项目，我们可以用 Dash 作出动态展示分数的界面。原理是：Dash 进程每经过一个时间间隔，就执行分数更新函数，该函数从数据库中获取分数，并将分数放到 python 的 list 里面，同时为了防止同一分数重复添加，在设计分数表时增加一个标志字段，如果添加到了 list 里面就改变这个字段，下一次获取这个字段的时候避开这些添加过的分数。

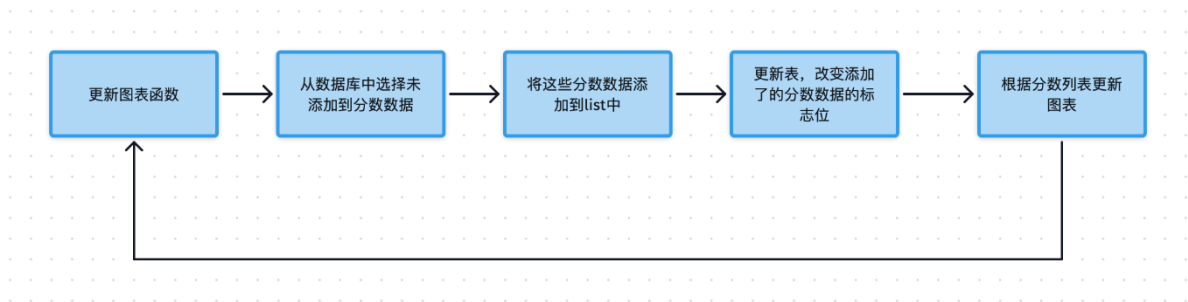


图 7 数据可视化的过程

出于好奇心, 我们选择使用 SQLite 作为 DBMS。使用 MySQL 还要装 connector 而且需要运行一个独立的进程, 配置麻烦。SQL 方便在 servless, 配置方便 (import 对应的包就可以了), 并且在这个项目中也不需要用到并发访问/用户管理的安全处理措施, SQLite 更适合开发和测试, 而且轻量。

5. 版本控制

在本次项目中, 受曹阳老师的启发, 我们尝试使用了 git 作为版本控制的工具, 我们将更新的代码 push 到 github 上面, 其他成员很方便地实时更新。git 有强大的对比功能和回滚功能, 方便我们查看版本之间的差异, 在遇到问题的时候方便回滚版本。

6. 人机对战界面效果

在设计人工对战界面效果的过程中, 本文主要将强化学习后的模型运行代码, 与未加强化学习的游戏原始代码, 分别作为两个包, 使用一个 pygame 窗口将它们整合起来。具体界面如下图 3 所示。人类控制方向只能影响未加强化学习的游戏界面小人的移动和跳跃。游戏开始后, 软件就会调用强化学习的模型, 自动选择最合适的板块跳跃。从而实现了人机博弈的效果。经测试, 人机每次只能对战一局, 对战体验良好。

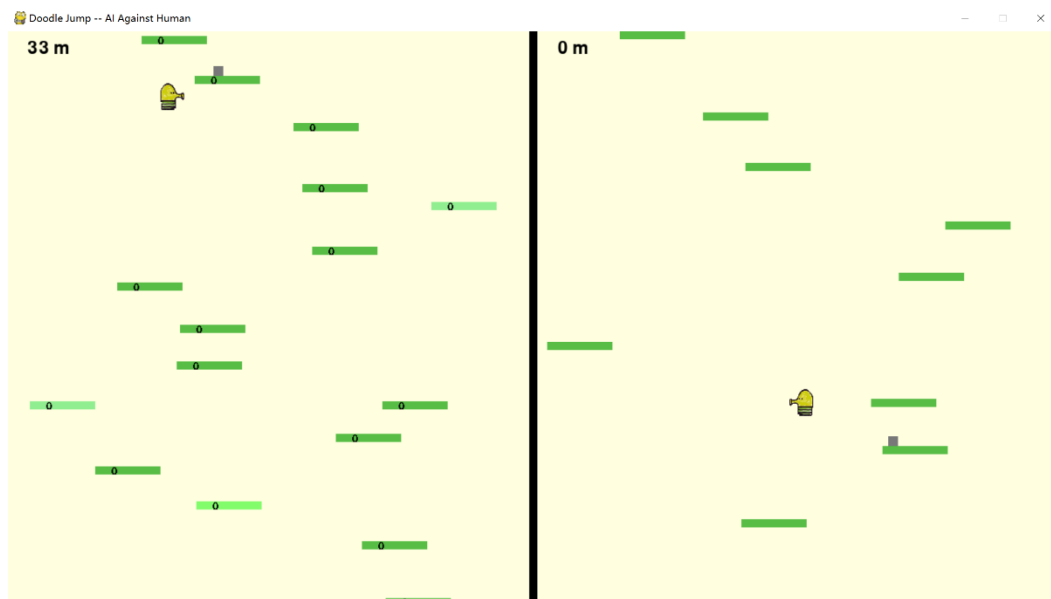


图 8 人机对战博弈效果

为了实现两种不同游戏模式的整合，需要对小组其他成员撰写的两种不同的代码进行整合，分别是包含训练模型的代码和未包含任何模型的原始游戏代码。将两种代码分别作为 python 的两个用户自定义包，分别剔除 `main.py` 和控制游戏的 `game.py`，以及修改固定的参数之后，重新编写 `main.py` 和 `game.py` 方便整合成为一个游戏界面。

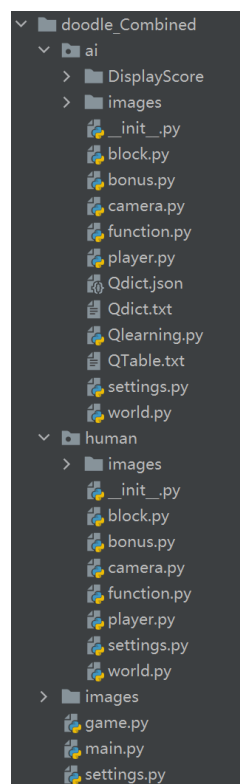


图 9 人机对战的文件结构图

四、实验结果分析

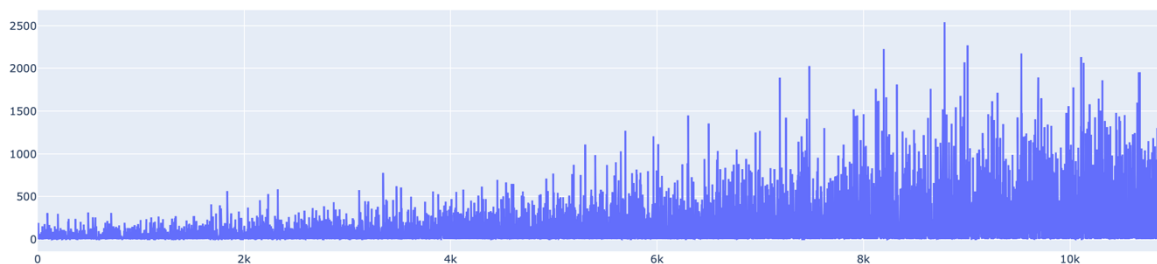


图 10 无压缩状态空间的训练数据折线图

(横坐标表示玩的局数，纵坐标表示智能体的得分)

训练中，我们取消弹簧。因为弹簧并不在智能体的考虑范围之内，而且弹簧的存在会干扰训练结果。当训练结束后，再将弹簧特殊处理即可。上图是进行空间压缩，运行 1 个半小时，游戏局数达 15k 的训练过程。可见 0~4k 轮时，玩家得分不高，最高分值达到 765 分。是因为智能体探索的状态过少，遇见陌生的情况概率较高，无法获取更高的分数。而当训练局数到达 4k 轮往后时，频繁出现得分超过 1000 分，可得智能体遇见的情况较多，能很好的应对大部分情况。但此时智能体仍会出现得分低的情况，可能是因为得分较低时，智能体所遇到的情况是陌生的，或者尚未修正好的。总体而言，智能体还是能很好的进行自我修正。相信随着训练时间的增长，智能体获取高分的情况会越多，会更为稳定。

五、结论

本次 Qlearning 的实现，总体上还是令人满意。虽说训练时间过长，但仍可以用许多方法去缩短训练时间。比如这次实现中采用的适当压缩 x 距离，还可以采用多线程训练，充分发挥多核 CPU 的性能。或者更新分数时，不仅是更新该状态的分数，还可以更新与此状态相近的状态的分数，与此状态越不相似，所更新的分数越小。由于大二下学业过于匆忙，部分方法未能在规定时间内实现，对此感到遗憾。

六、参考文献

- [1] 薛俏, 丁慧霞, 张庚, 等. 基于 Q Learning 算法的电力通信业务路由规划[J]. 光学与光电技术, 2019, 17(4): 51-56.
- [2] Watkins C J C H, Dayan P. Q-learning[J]. Machine learning, 1992, 8(3-4): 279-292.