深圳大学考试答题纸

(以论文、报告等形式考核专用)

二〇二〇～二〇二一　学年度第 二 学期

课程编号	1504440001	课序号	01	课程名称	Python 程序设计	主讲教师	潘浩源	评分	
学　　号	2019092121	姓名	沈晨玓	专业年级	19 级计算机科学与技术				
学　　号	2019284073	姓名	董芸豪	专业年级	19 级计算机科学与技术				
学　　号	2019151099	姓名	肖又旗	专业年级	19 级软件工程				

教师评语:

题目:　基于强化学习与深度强化学习的游戏 AI 训练

一、摘要

在本次大作业中由两个项目组成。

第一个小项目即为简单 AI 走迷宫游戏，通过**强化学习的 Q-learning 算法**，对 AI 进行训练来让其能以大概率找打一条通关路径并基本按照该路径进行移动。第二个小项目基于 Gym 库提供的 Atari 游戏 Pong，通过**深度强化学习的 DQN 算法**，对 AI 进行训练来让其能与机器进行弹球对战。

二、引言

第一个项目通过利用强化学习中的 Q-learning 算法，实现了游戏 AI 去游玩走迷宫游戏并且找到最大概率通过迷宫的目标。这个项目主要利用了强化学习的入门知识去实现简单的走迷宫游戏例子，并通过训练我们自己的 AI 去从重重障碍中走出迷宫。

第二个项目通过利用 DQN 算法，实现了 Pong 弹球小游戏中的 AI 训练。我们训练我们自己的 AI 完成 Pong 游戏。经过 DQN 算法对 AI 的训练，可以看到我们的 AI 胜率有了很大改善，从最开始几乎全输到几乎全胜。

三、研究方法

（一）项目一：Q-Learning 算法在 AI 走迷宫游戏中的探索

1、Q-learning 算法介绍

1.1. 什么是 Q-learning：

Q-learning 是一种用于机器学习的强化学习技术。 Q-learning 的目标是学习一种策略，告诉 Agent 在什么情况下要采取什么行动。 它不需要环境模型，可以处理随机转换和奖励的问题。

对于任何有限马尔可夫决策过程（FMDP），Q 学习找到一种最优的策略，即从当前状态开始，它在任何和所有后续步骤中最大化总奖励的预期值。在给定无限探索时间和部分随机策略的情况下，Q 学习可以为任何给定的 FMDP 确定最佳动作选择策略。“Q”命名函数返回用于提供强化的奖励，并且可以说代表在给定状态下采取的动作的“质量”。

1.2. 强化学习(Reinforcement learning)

强化学习涉及一个代理(Agent)，一组状态 S(state)，以及一组动作 A(action)。通过在状态中执行动作从而切换到另一个状态。在特定状态下执行动作为 Agent 提供奖励

(reward) Agent 的目标是最大化其总（未来）奖励。 它通过将未来状态可获得的最大奖励添加到实现其当前状态的奖励来实现这一点，从而通过潜在的未来奖励有效地影响当前行动。 该潜在奖励（价值）是从当前状态开始的所有未来步骤的奖励的预期值的加权总和。

例如：考虑登上火车的过程，其中奖励是通过登上火车总时间的负值来衡量的（值越小代表登上火车时间越短，这是更优的选择）。 一种策略是一旦打开就进入火车门，最大限度地缩短了自己的初始等待时间。 然而，如果火车很拥挤，那么在你进入大门的最初动作之后你将会进入缓慢状态，因为当你试图登上火车时，其他上火车的人会将你推离火车。 总的登上火车时间（成本）是：

0 秒等待时间+ 15 秒上车时间

第二天，通过随机机会（探索），你决定等待并让其他人先离开。 这最初导致更长的等待时间。 但是，由于其他上车人竞争，于是你可以更快的登上火车。 总体而言，此路径的奖励高于前一天，因为总的登上火车时间现在是：

5 秒等待时间+ 0 秒上车时间

通过探索，尽管最初行动导致更大的成本（或负面奖励），而不是强有力的策略，总体成本更低，从而揭示了更有价值的策略。

2、Q 表(Q-table)

在训练过程中需要对游戏中遇到的场景的一些选择情况进行保存，因此可以建立如下的表

Q-table	Action1	Action2	Action3	Action4
State1	Q (s1, a1)	Q (s1, a2)	Q (s1, a3)	Q (s1, a4)
State2	Q (s2, a1)	Q (s2, a2)	Q (s2, a3)	Q (s2, a4)
State3	Q (s3, a1)	Q (s3, a2)	Q (s3, a3)	Q (s3, a4)

如上表所示，Q 表中保存了在游戏进程中所将遇到的若干个场景，以及在场景中 Agent 将会遇到的所有情况下所能做出的相应应对动作选择（离散的），而每一个 Q 值所代表的就是在当前环境状态下，所选择了某个动作来应对之后在游戏结束时所 Agent 期望所能得到的分数。

由于我们所做的走迷宫 demo 的环境状态和对应的动作都是**离散的**，因此可以利用 Q 表来保存每轮训练的结果。

3、公式推导：

正如引言中所提到的，本子项目的主要目的是训练一个能够玩走迷宫游戏的 AI 并且在该游戏中拿到高分，因此我们的目标就是求出累计奖励最大的策略的期望：

$$Goal = max_{\pi} E[\sum_{t=0}^{H} \gamma^{k} R(S_t A_t S_{t+1}) | \pi]$$

Q-learning 的主要优势就是使用了时间差分法 TD（融合了蒙特卡洛和动态规划）能够进行离线学习，使用 bellman 方程可以对马尔科夫过程求解最优策略。

通过 bellman 方程求解马尔科夫决策过程的最佳决策序列，状态值函数$V_\pi(s)$可以评价当前状态的好坏，每个状态的值不仅仅由当前状态决定，还需要由后面的状态决定，所以状态的累计奖励期望就可以得出当前s的状态值函数$V(s)$。

$$V_\pi(s) = E(U_t|S_t = s)$$

$$V_\pi(s) = E_\pi\left[R_{t+1} + \gamma\left[R_{t+2} + \gamma[\dots\dots]\right]|S_t = s\right]$$

$$V_\pi(s) = E_\pi[R_{t+1} + \gamma V(s')|S_t = s]$$

最优累计期望可用$V^*(s)$表示，可知最优解函数就是$V^*(s) = max_\pi V_\pi(s)$

$$V^*(s) = max_\pi E[\sum_{t=0}^{H} \gamma^k R(S_t A_t S_{t+1})|\pi, S_0 = S]$$

$Q(s,a)$状态动作值函数

$$q_\pi(s,a) = E_\pi[r_{t+1} + \gamma r_{t+2} + r^2\gamma_{t+3} + \cdots|A_t = a, S_t = s]$$

$$q_\pi(s,a) = E_\pi[G_t|A_t = a, S_t = s]$$

其中G_t是t时刻开始的总折扣奖励，从这里我们能看出来 γ衰变值对Q函数的影响，γ越接近于1代表它越有远见会着重考虑后续状态的价值，当γ接近0的时候就会变得近视只考虑当前的利益的影响。所以从 0 到1，算法就会越来越会考虑后续回报的影响。

最优价值动作函数$Q^*(s,a) = max_\pi Q^*(s,a)$

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)(R(s,a,s') + \gamma max_{a'} Q^*(s',a'))$$

Bellman 方程实际上就是价值动作函数的转换关系

$$V_\pi(s) = \sum_{a\in A} \pi(a|s) q_\pi(s,a)$$

$$q_\pi(s,a) = R_s^a + \gamma \sum_{s'\in S} P_{ss'}^a V_\pi(s')$$

$$V_\pi(s) = \sum_{a\in A} \pi(a|s)\left[R_s^a + \gamma \sum_{s'} P_{ss'}^a V_\pi(s')\right]$$

4、更新公式以及伪代码实现

根据以上推导可以对Q值进行计算，所以有了Q值我们就可以进行学习，也就是 Q-table 的更新过程，其中α为学习率γ为奖励性衰变系数，采用时间差分法的方法进行更新。

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$$

上式就是 Q-learning 更新的公式，根据下一个状态s'中选取最大的$Q(s',a')$值乘以衰变γ加上真实回报值最为Q现实，而根据过往Q表里面的$Q(s,a)$作为Q估计。

整个训练程序的伪代码如下：

```
1. Initialize Q arbitrarily //随机初始化 Q 值
2. Repeat (for each episode): //每一次游戏，从 agent 出发到游戏结束是一个
   episode
```

```
    Initialize S //agent 处于初始位置(左上角)，S 为初始位置的状态
    Repeat (for each step of episode):
        根据当前 Q 和位置 S，使用一种策略，得到动作 A //这个策略可以是 ε-greedy 等
        做了动作 A，agent 到达新的位置 S'，并获得奖励 R //奖励可以是 1，50 或者-1000
        Q(S,A) ← (1-α)*Q(S,A) + α*[R + γ* max Q(S', a)] //在 Q 中更新 S
        S ← S'
until S is terminal //即到规定训练轮数为止
```

5、程序主要代码

①从 Q 表中选取动作：

```python
# 从 Q-table 中选取动作
def get_action(self, state):
    if np.random.rand() < self.epsilon:
        # 贪婪策略随机探索动作
        action = np.random.choice(self.actions)
    else:
        # 从 q 表中选择
        state_action = self.q_table[state]
        action = self.arg_max(state_action)
    return action
```

②采样 <s, a, r, s'>

```python
# 采样 <s, a, r, s'>
    def learn(self, state, action, reward, next_state):
        current_q = self.q_table[state][action]
        # 贝尔曼方程更新
        new_q = reward + self.discount_factor * max(self.q_table[next_state])
        self.q_table[state][action] += self.learning_rate * (new_q - current_q)
```

③agent 从 Q 表中选择出最优动作

```python
#agent 如何在 Q 表中选择出最优动作
@staticmethod
    def arg_max(state_action):
        max_index_list = []
        max_value = state_action[0]
        for index, value in enumerate(state_action):
            if value > max_value:
```

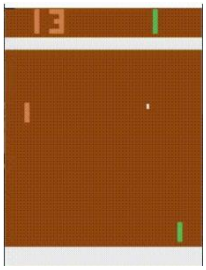
```
                max_index_list.clear()
                max_value = value
                max_index_list.append(index)
            elif value == max_value:
                max_index_list.append(index)
        return random.choice(max_index_list)
```

（二）项目二：DQN 算法在弹球游戏中的探索

1、DQN 算法介绍

上一部分介绍关于 Q-Learning 的算法原理及 DEMO 实现流程，可以看出 Q-Learning 算法通过建立 Q 表完成了对于简单游戏的训练。在上面的简单分析中，我们使用表格来表示$Q(s,a)$，但是这个在现实的很多问题上是几乎不可行的，因为状态实在是太多。使用表格的方式根本存不下，会造成维度爆炸问题。

以我们即将完成的 Demo---Atari 中的 pong 游戏为例：



计算机玩 Atari 游戏的要求是输入原始图像数据，也就是 210x160 像素的图片，然后输出几个按键动作。总之就是和人类的要求一样，纯视觉输入，然后让计算机自己玩游戏。那么这种情况下，到底有多少种状态呢？有可能每一帧的状态都不一样。因此，从理论上看，如果每一个像素都有 256 种选择，那么就有：$256^{210\times160}$ 种状态。所以，我们是不可能通过表格来存储状态的。我们有必要对状态的维度进行压缩。

2、DQN 算法原理解释

①价值函数近似

说起来很简单，就是用一个函数来表示$\mathrm{Q(s,a)}$，即$\mathrm{Q(s,a)=f(s,a)}$。f可以是任意类型的函数$\mathrm{Q(s,a)=w_1s+w_2a+b}$，其中$\mathrm{w_1,w_2,b}$是函数$f$的参数。

通过函数表示，我们就可以无论s到底是多大的维度，最后都通过矩阵运算降维输出为单值的Q。因为我们并不知道Q值的实际分布情况，本质上就是用一个函数来近似Q值的分布，用w来统一表示函数f的参数，我们就可以得到$\mathrm{Q(s,a)\approx f(s,a,w)}$。

对于 Atari 游戏而言，这是一个高维状态输入（原始图像），低维动作输出（上下左右）。我们只需要对高维状态进行降维，而不需要对动作也进行降维处理。

处理方法是只把状态 s 作为输入，但是输出的时候输出每一个动作的 Q 值，也就是输出一个向量$[Q(s,a_1),Q(s,a_2),\ldots,Q(s,a_n),]$，这里输出是一个值，只不过是包含了所有动作的 Q 值的向量而已。这样我们就只要输入状态 s，而且还同时可以得到所有的动作 Q 值，也将更方便的进行 Q-Learning 中动作的选择与 Q 值更新。

②Q 值神经网络化

我们用一个深度神经网络来表示这个函数f。

输入是经过处理的 4 个连续的 84x84 图像（处理方式是利用[1]提到的预处理方法）经过三个卷积层，两个全连接层，最后输出包含每一个动作 Q 值的向量。

③DQN 算法

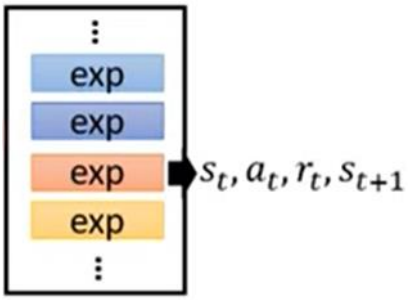
神经网络的训练是一个最优化问题，最优化一个损失函数，目标是让损失函数最小化。基于 Q-Learnning 算法，我们把目标 Q 值作为标签，目标就是让 Q 值趋近于目标 Q 值。因此，DQN 训练的损失函数就是$L(w)=E[(r+\gamma maxQ(s',a',w)-Q(s,a,w))^2]$。确定了损失函数，确定了获取样本的方式。那么 DQN 的整个算法也就成型了。

④DQN 训练

通过 Q-Learning 获取无限量的训练样本，然后对神经网络进行训练。但是仅仅通过简单的梯度下降算法 SGD，对于网络的训练效果较差，因此引入了一下两个概念：经验回放与目标网络。

（1）经验回放

经验池的功能主要是解决相关性及非静态分布问题。具体做法是把每个时间戳 Agent 与环境交互得到的转移样本(s_t,a_t,r_t,s_{t+1})储存到回放记忆单元，要训练时就随机拿出一些（minibatch）来训练。（其实就是将游戏的过程打成碎片存储，训练时随机抽取就避免了相关性问题）。



这样做可以减少经验浪费并减轻经验关联性对网络训练的影响

伪代码：NIPS 2013 DQN 算法

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

（2）目标网络

在[2]Nature 2015 版本的 DQN 中提出了这个改进，使用另一个网络（这里称为 target_net）产生 Target Q 值。具体地，$Q(s,a,w_t)$表示当前网络 eval_net 的输出，用来评估当前状态动作对的值函数；$Q(s,a,w_{t+1})$表示 target_net 的输出，代入上面求 Target Q 值的公式中得到目标 Q 值。根据上面的 Loss Function 更新 eval_net 的参数，每经过 N 轮迭代，将参数复制给 target_net。

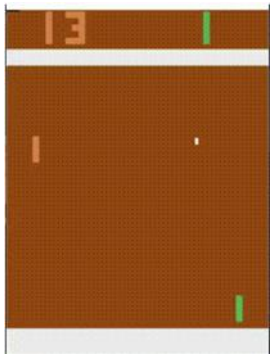
引入 target_net 后，再一段时间里目标 Q 值使保持不变的，一定程度降低了当前 Q 值和目标 Q 值的相关性，提高了算法稳定性。

伪代码：Nature 2015 DQN 算法

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

3、Demo 实现流程

①项目架构：
基于 Pytorch 框架进行开发，完成了对于 Atari 游戏 Pong 的训练。



主要文件：
Env.memory.py：经验池
Env.models.py：DQN 神经网络搭建
Env.wrappers.py：图像预处理
Dqn.py：主程序入口

数据分析：
Pong_run：tensorboard 文件
Log/DQN：训练轮数与得分记录

②训练参数

```
parser = argparse.ArgumentParser()
parser.add_argument('--device', type=str,
                    default='cuda' if torch.cuda.is_available() else 'cpu')
parser.add_argument('--batch-size', type=int, default=32)
parser.add_argument('--seed', type=int, default=0)
parser.add_argument('--gamma', type=float, default=0.99)
parser.add_argument('--eps-start', type=float, default=0.8)
parser.add_argument('--eps-end', type=float, default=0.02)
parser.add_argument('--eps-decay', type=int, default=1000000)
parser.add_argument('--target-update', type=int, default=1000)
parser.add_argument('--lr', type=float, default=0.0001)
parser.add_argument('--initial-memory', type=int, default=500)
parser.add_argument('--train-episodes', type=int, default=1600)
parser.add_argument('--render-train', default=False,
                    action="store_true", help="Render the Training Process")
parser.add_argument('--render-test', default=False,
                    action="store_true", help="Render the Testing Process")
args = parser.parse_known_args()[0]
args.memory_size = 200000
```

③DQN 整体思路

```
def train(args, env, n_episodes, render=True):
    render = True
    writer_path = os.path.join('pong_runs', 'dqn_' + args.a_path)
    writer = SummaryWriter(writer_path)
    episode_list = []
    reward_list = []
    for episode in range(n_episodes):
        obs = env.reset()
        state = get_state(obs)
        total_reward = 0.0
        for t in count():

            action = select_action(args, state)
            obs, reward, done, info = env.step(action)

            if render:
                env.render()

            total_reward += reward

            if not done:
                next_state = get_state(obs)
            else:
                next_state = None

            reward = torch.tensor([reward], device=args.device)

            args.memory.push(state, action.to('cpu'),
                             next_state, reward.to('cpu'))
            state = next_state

            if args.steps_done > args.initial_memory:
                optimize_model(args)
                if args.steps_done % args.target_update == 0:
                    args.target_net.load_state_dict(
                        args.policy_net.state_dict())

            if done:
                episode_list.append(episode)
                reward_list.append(total_reward)
                break

        # 写入 tensorboard 用于看图
        writer.add_scalar("reward", total_reward, episode)
        # 将每局对应的总奖励写成 txt 文件，用于 matplotlib 画图
```

```
        write_file(args.path+'/episodes.txt', episode_list)
        write_file(args.path+'/rewards.txt', reward_list)

        if episode % 10 == 0 or episode == n_episodes-1:
            print('Total steps: {} \t Episode/t: {}/{} \t Total reward: {}'.format(
                args.steps_done, episode, t, total_reward))
    env.close()
    return
```

④经验池

```
class ReplayMemory(object):
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def push(self, *args):
        if len(self.memory) < self.capacity:
            self.memory.append(None)
        self.memory[self.position] = Transition(*args)
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

⑤神经网络搭建

```
class DQN(nn.Module):
    def __init__(self, in_channels=4, n_actions=14):
        super(DQN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, 32, kernel_size=8, stride=4) # 4*84*84-->32*20*20
        self.conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)# 32*20*20-->64*9*9
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)# 64*9*9-->64*7*7

        self.fc4 = nn.Linear(7 * 7 * 64, 512)
        self.head = nn.Linear(512, n_actions)

```

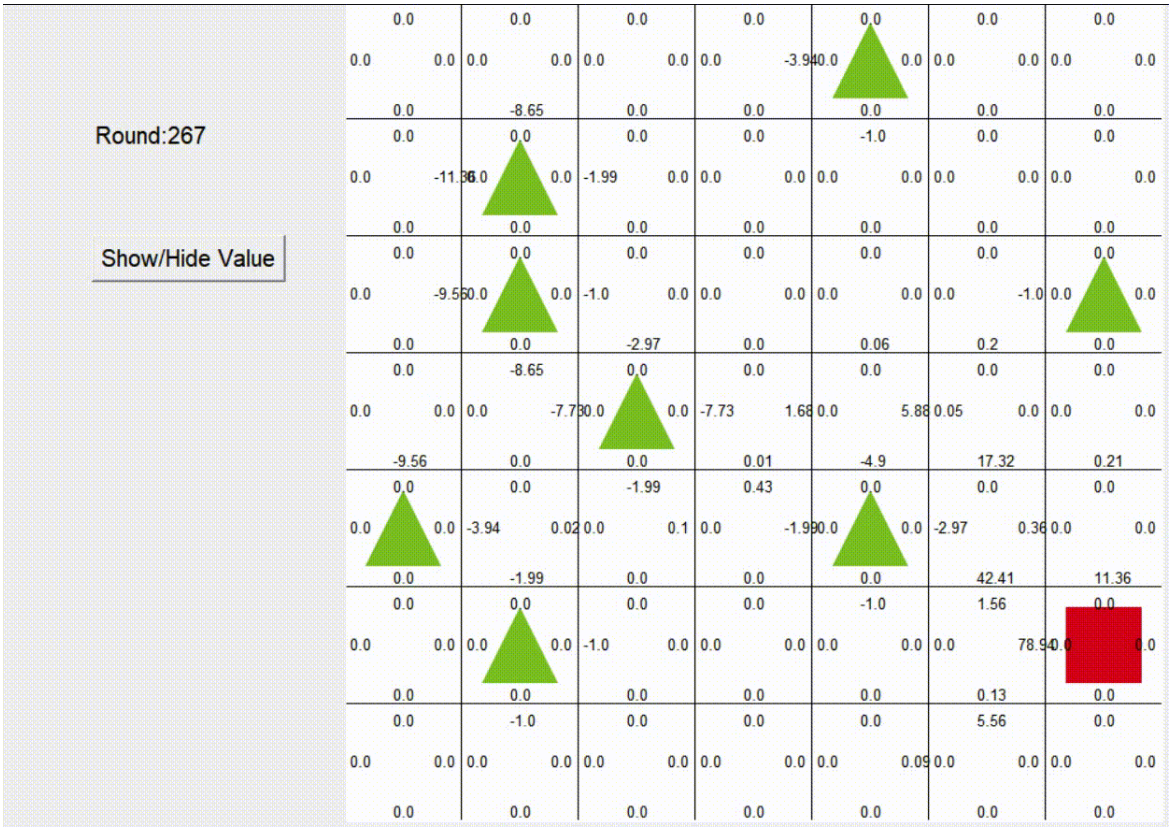
```
    def forward(self, x):
        x = x.float() # 255
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))

        x = F.relu(self.fc4(x.view(x.size(0), -1)))
        return self.head(x)
```

四、研究结果与项目结论

1、项目一

为了更好的展示训练的结果，我们设置了一个情景，即小人走迷宫。



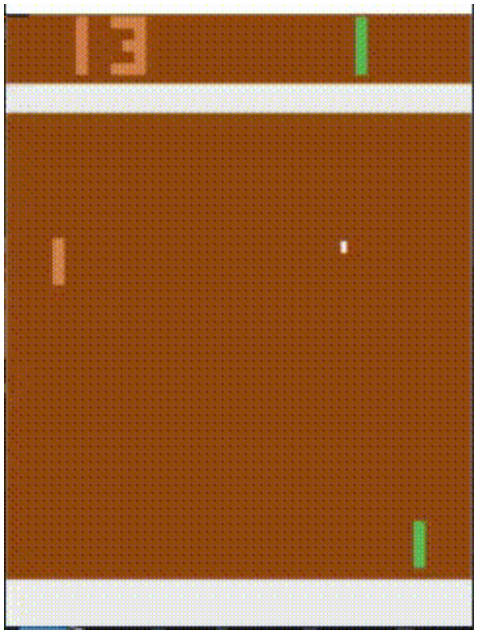
在上图的迷宫中，红色方框代表小人（我们的 agent），绿色三角为迷宫中的障碍物，小人（我们的 agent）碰到就会死亡，蓝色圆圈代表迷宫的终点，到达即获胜。

程序最开始运行时小人会经常撞到绿色三角障碍物上；300 轮训练后，大致可以找到走出迷宫的路径。1000 轮训练结束后，基本上已经找到了走出迷宫的路径。

此外，也可以见地图上每个格子的四个方位上都存在有一个数值，该数值就是我们的 Q 值，值越大 agent 在这个位置时选择走这个方向的概率就更大。经过 1000 轮训练后我们可以发现，存在一条路径中 Q 值远大于其他路径 Q 值的路，这即为找到的走出迷宫的路。

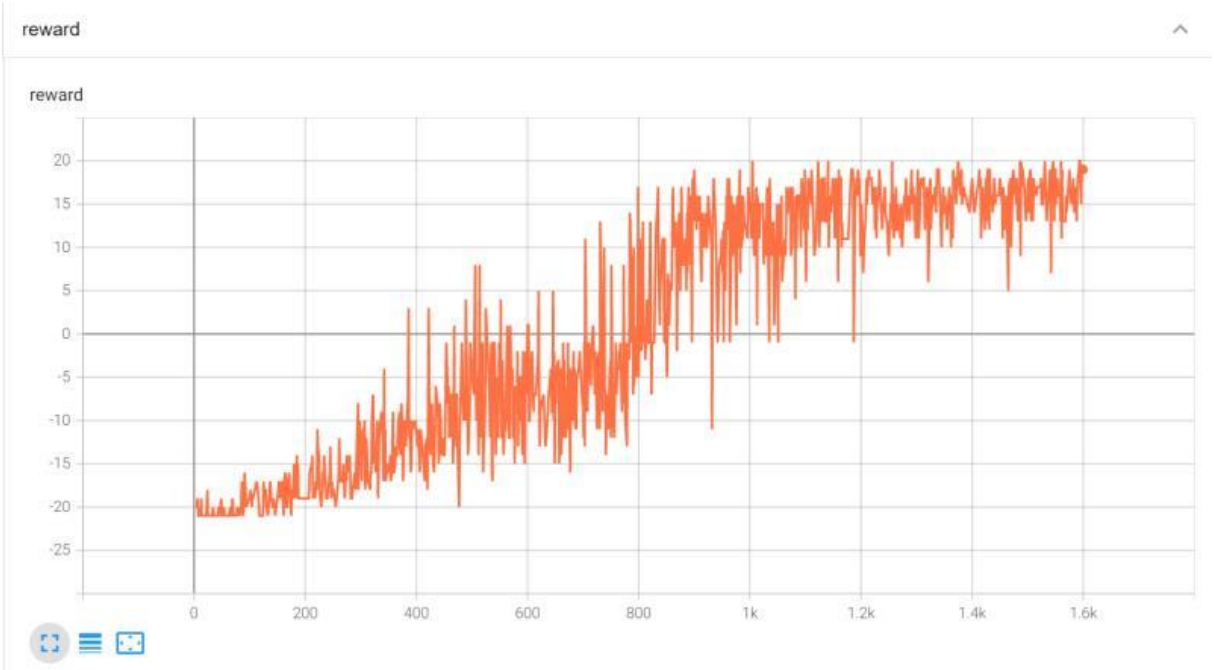
2、项目二

我们使用弹球游戏对训练结果进行测试。

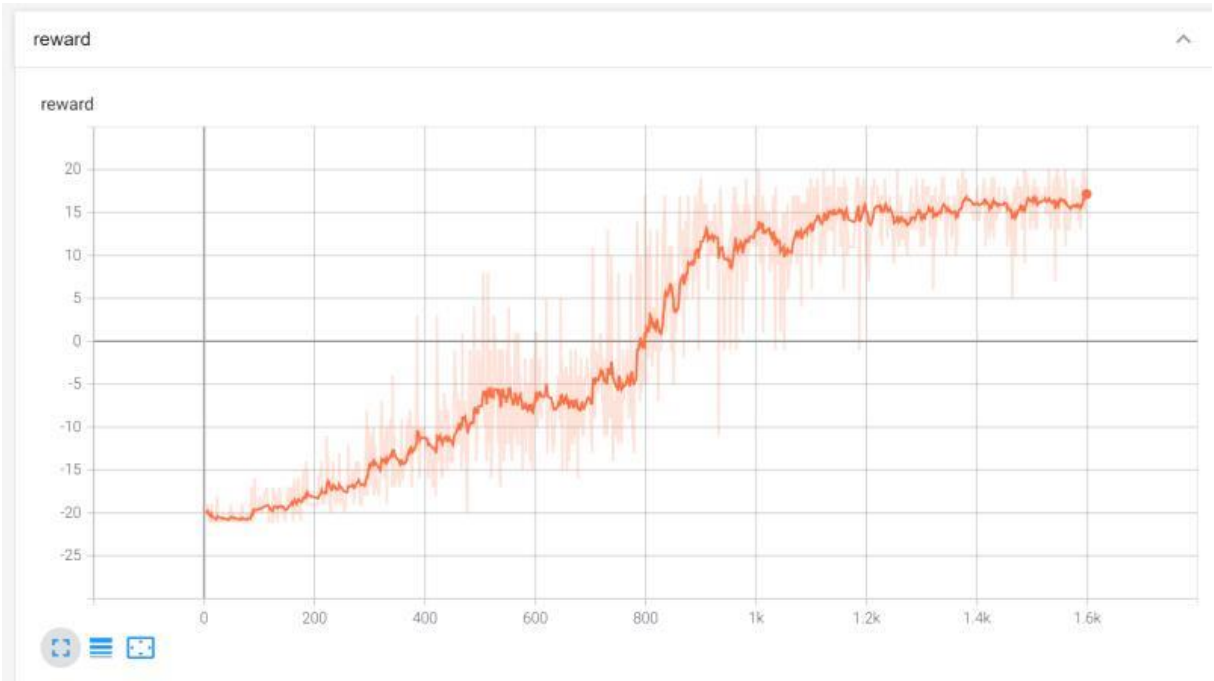


我们控制右侧的绿色板子，电脑控制另一块橙色板子。小球从屏幕中央弹出，通过移动一侧板子将小球弹给对方。当一方未接到由另一方弹来的小球，即判负，为对手加上一分。我们设定当一方达到 20 分时为一轮。进行多轮次的训练。由于训练过程时间消耗很大，我们使用 Tensorboard 进行训练结果的可视化。当训练刚开始时，绿方完全不能接住弹出的小球，前几轮几乎橙色方得 20 分，绿方得 0 分。训练运行一段时间后，橙色方得分仍为 20 分，但绿色方得分变多。训练再运行一段时间后，逐渐打平。运行很长时间后，橙色方几乎不得分，绿色方可以完美接到橙色方弹出的小球。

利用 Tensorboard 对训练结果进行可视化，可得到如下曲线。



对曲线进行平滑处理后，可以得到如下训练曲线：



可以看出，随着训练时间的增长，橙色方得分逐渐增加，当训练到第 800 轮时基本打平。当训练到第 1600 轮时，橙色方几乎不得分。

五、致谢与参考文献

一学期的课程很快就画上了句号，我们也顺利地完成了所有老师布置的作业以及期末项目。

首先感谢潘浩源老师一学期的教学，教会了我们许多 Python 编程的知识。16 节精心准备的线下课清晰地为我们讲解了 Python 的基础知识，从基础知识到数据结构，从函数式编程到面向对象编程，从 numpy 到标准库，这些精心准备的课程为我们打开了一扇 Python 的大门，让习惯了冗长 C++与 Java 的我们震撼与 Python 方便的库函数以及灵活的编程模式。9 次难度适中，题目经典的实验让我们巩固了课上所学，在实践中体会 Python 的魅力。2 次作业，让我们从比较实际问题入手，学习了 Python 解决问题的思想。1 次大作业，让我们自己拓展迁移，自行学习自行接触陌生领域给我们一把探索的钥匙去发掘 Python 之宝。再次感谢潘老师一学期的精心备课，上课的清晰讲解以及课后认真负责的作业选题与作业批改。再次感谢潘老师一学期的付出！

其次需要感谢的是张昊迪老师以及陈振浩师兄。本次期末项目是实验室项目的前期学习成果，作为深度学习与强化学习领域的小白，张老师以及师兄提供了许多帮助，解答了许多学术上的困扰。实验初期，由于种种原因神经网络的搭建与训练成果并不理想，正在师兄的帮助下，我们才得以成功地完成了神经网络的训练并做出了满意的成果。特此感谢张昊迪老师以及陈振浩师兄的帮助。

最后希望《Python 程序设计》这门课可以越办越好！

参考文献：

项目一：

[1] https://mofanpy.com/tutorials/machine-learning/reinforcement-learning/tabular-q2/

[2] https://www.bilibili.com/video/BV13W411Y75P?p=8

[3] https://blog.csdn.net/sinat_32485497/article/details/75313499

[4] https://blog.csdn.net/qq_30615903/article/details/80739243

项目二：

[1] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

[2] Mnih V., Kavukcuoglu K., Silver D., et al. Human-level control through deep reinforcement learning. Nature,2015,518(7540):529-533.

[3] Haodi Zhang, Zihang Gao, Yi Zhou, et al. Faster and Safer Training by Embedding High-Level Knowledge into Deep Reinforcement Learning. arXiv preprint. 2019:1910.09986.

[4] Shusen Wang DQN 高级技巧 https://www.youtube.com/watch?v=rhslMPmj7SY&list=PLvOO0btloRntS5U8rQWT9mHFcUdYOUmIC

[5] Hung-yi Lee DRL Lecture https://www.youtube.com/watch?v=o_g9JUMw1Oc

[6] 莫凡 Python 强化学习 https://mofanpy.com/tutorials/machine-learning/reinforcement-learning/

[7] Shusen Wang CS583：DeepLearning https://github.com/wangshusen/DeepLearning