Mid-term Assignments

【院系】计算机学院

【专业】计算机科学与技术

【学号1】20337234

【姓名1】宋丽铭

【学号2】20337066

【姓名2】李亚茹

开源仓库链接

【简要介绍】

Atari Breakout是一款由雅达利开发及发布的打砖块的街机游戏，控制木板左右移动把球反弹到上面来消除砖块。

一共8排砖块，每两排砖块的颜色不同。由下至上的颜色排序为黄色、绿色、橙色和红色。游戏开始后，玩家必须控制一块平台左右移动以反弹一个球。当那个球碰到砖块时，砖块就会消失，而球就会反弹。当玩家未能用平台反弹球的话，那么玩家就输掉了那个回合。当玩家连续输掉3次后(这里使用了5次)，玩家就会输掉整个游戏。玩家在游戏中的目的就是清除所有砖块。

【原理】

1 强化学习

1.1 什么是强化学习?

> 强化学习（Reinforcement Learning, RL），又称再励学习、评价学习或增强学习，是机器学习的范式和方法论之一，用于描述和解决智能体（agent）在与环境的交互过程中通过学习策略以达成回报最大化或实现特定目标的问题。

1.2 强化学习的组成部分

- Agent（智能体、机器人、代理）：强化学习训练的主体就是Agent，有时候翻译为“代理”，这里统称为“智能体”。
- Environment（环境）：整个游戏的大背景就是环境
- State（状态）：当前 Environment和Agent所处的状态
- Action（行动）：基于当前的State，Agent可以采取哪些action，比如向左or右，向上or下；Action是和State强挂钩的，比如上图中很多位置都是有隔板的，很明显Agent在此State下是不能往左或者往右的，只能上下；
- Reward（奖励）：Agent在当前State下，采取了某个特定的action后，会获得环境的一定反馈就是Reward。这里面用Reward进行统称，虽然Reward翻译成中文是“奖励”的意思，但其实强化学习中Reward只是代表环境给予的“反馈”，可能是奖励也可能是惩罚。

1.3 强化学习的主要特点

试错学习：强化学习需要训练对象不停地和环境进行交互，通过试错的方式去总结出每一步的最佳行为决策，整个过程没有任何的指导，只有冰冷的反馈。所有的学习基于环境反馈，训练对象去调整自己的行为决策。

延迟反馈：强化学习训练过程中，训练对象的“试错”行为获得环境的反馈，有时候可能需要等到整个训练结束以后才会得到一个反馈，比如Game Over或者是Win。当然这种情况，我们在训练时候一般都是进行拆解的，尽量将反馈分解到每一步。

时间是强化学习的一个重要因素：强化学习的一系列环境状态的变化和环境反馈等都是和时间强挂钩，整个强化学习的训练过程是一个随着时间变化，而状态&反馈也在不停变化的，所以时间是强化学习的一个重要因素。

当前的行为影响后续接收到的数据：为什么单独把该特点提出来，也是为了和监督学习&半监督学习进行区分。在监督学习&半监督学习中，每条训练数据都是独立的，相互之间没有任何关联。但是强化学习中并不是这样，当前状态以及采取的行动，将会影响下一步接收到的状态。数据与数据之间存在一定的关联性。

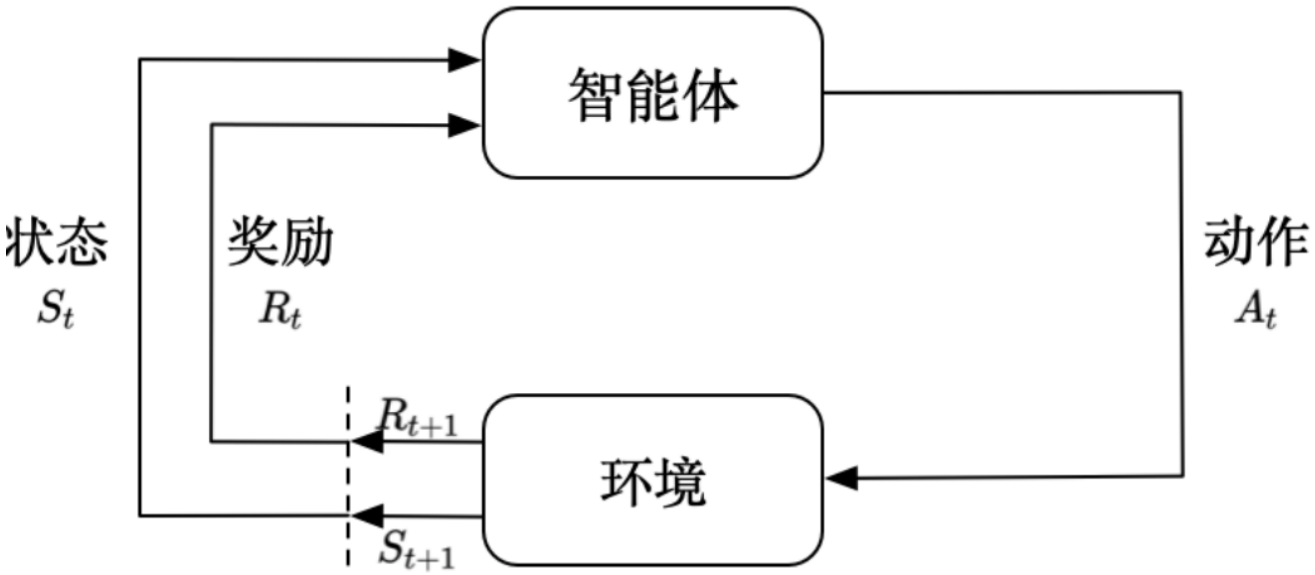


图 1.1　强化学习示意图

2 Q-learning

Q-learning 算法是一种value-based的强化学习算法，Q是quality的缩写，Q函数 Q(state，action)表示在状态state下执行动作action的quality， 也就是能获得的Q-value是多少。算法的目标是最大化Q值，通过在状态state下所有可能的动作中选择最好的动作来达到最大化期望reward。

Q-learning算法使用Q-table来记录不同状态下不同动作的预估Q值。在探索环境之前，Q-table会被随机初始化，当agent在环境中探索的时候，它会用贝尔曼方程（Bellman equation）来迭代更新Q(s,a)， 随着迭代次数的增多，agent会对环境越来越了解，Q函数也能被拟合得越来越好，直到收敛或者到达设定的迭代结束次数。

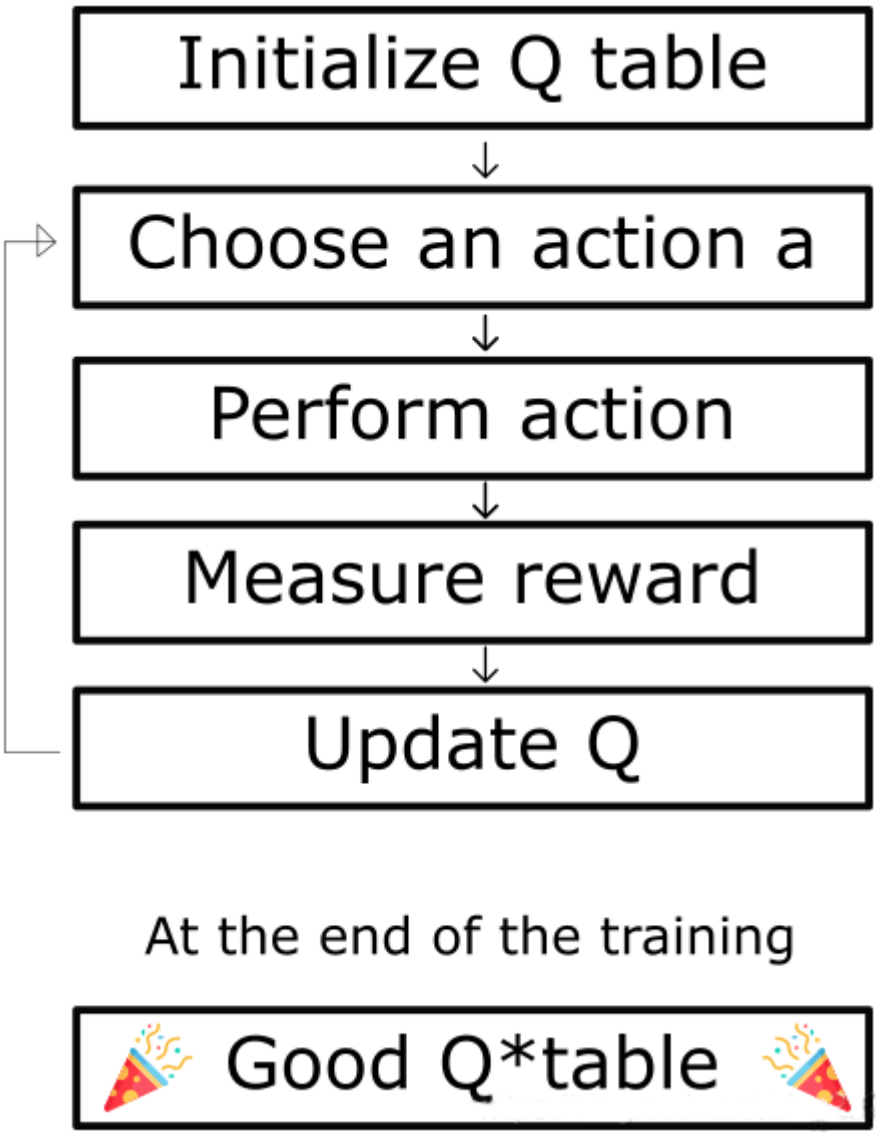
更新公式：

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。

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

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。

Q-learning算法流程：



3 DQN（Deep Q Network）

3.1 为什么需要DQN

Q-learning的核心在于Q表格，通过建立Q表格来为行动提供指引，但这适用于状态和动作空间是离散且维数不高时，当状态和动作空间是高维连续时Q表格将变得十分巨大，对于维护Q表格和查找都是不现实的。设想一下如果AlphaGo使用Q-learning将会是什么样的场景，围棋的可能性量级为10^170，如此巨大的Q表格已经丧失了的它的价值。

Q表格无法解决，因此，考虑一种值函数近似的方法，实现每次只需事先知晓S或者A，就可以实时得到其对应的Q值。深度学习在复杂特征提取效果良好，所以DQN采用了深度神经网络作为值函数近似的工具，将RL与DL结合变得到了DQN。

这将带来两个好处：

1.只需要存储DL的网络结构与参数 2.相近的的输入会得到相近的输出，泛化能力更强

3.2 DQN（NIPS 2013）

Q-learning算法很早就有了，但是其与深度学习的结合是在2013年的DeepMind发布的《Playing Atari with Deep Reinforcement Learning》论文中才实现的。这篇论文创造性的将RL与DL实现了融合，提出了存储记忆（Experience Replay）机制和目标网络（Fixed-Q-Target），实现了一部分Atari游戏操控，甚至超过了人类水平。这篇论文的创新点，就是刚刚提到的 Experience Replay 和 Fixed-Q-Target。

Experience Replay 经验回放

经验池回放，我们将agent在每个时间步骤的经验储存在数据中，将许多回合汇聚到一个回放内存中，数据集$D = e_1, .., e_N$，其中$e_t = (s_t, a_t, r_t, s_{t+1})$。在算法的内部循环中，我们会把从部分数据中进行随机抽样，将抽取的样本作为神经网络的输入，从而更新神经网络的参数。使用经验回放的优势有：

- 经验的每个步骤都可能在许多权重更新中使用，这会提高数据的使用效率；
- 在游戏中，每个样本之间的相关性比较强，相邻样本并不满足独立的前提。机器从连续样本中学习到的东西是无效的。采用经验回放相当于给样本增添了随机性，而随机性会破坏这些相关性，因此会减少更新的方差。

Fixed-Q-Target 目标网络

该论文的另一个创新点在于没有只采用一个神经网络进行训练，而是设计了两个结构完全相同的神经网络，分别为目标网络（Q-target）和评估网络（Q-predict），但Q-target网络中采用的参数是旧参数，而Q-predict网络中采用的参数是新参数。在训练过程中，Q-predict的参数不断地更新优化，只在一定间隔时间点将评估网络的参数赋给Q-target。Q-predict根据输入的s状态给出预测的q_{value}，Q-target根据s_{next}的输入给出q_{next}，再将q_{next}代入Q-learning的核心递推式中可以得到q_{target}，最后 $loss(q_{value}, q_{target})$ 反向传播并优化参数即可不断地得到更好的神经网络，这样经过反复的训练优化后，智能体基本可以掌握各种状态下的action决策。

DQN伪代码：

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

3.3 Nature DQN

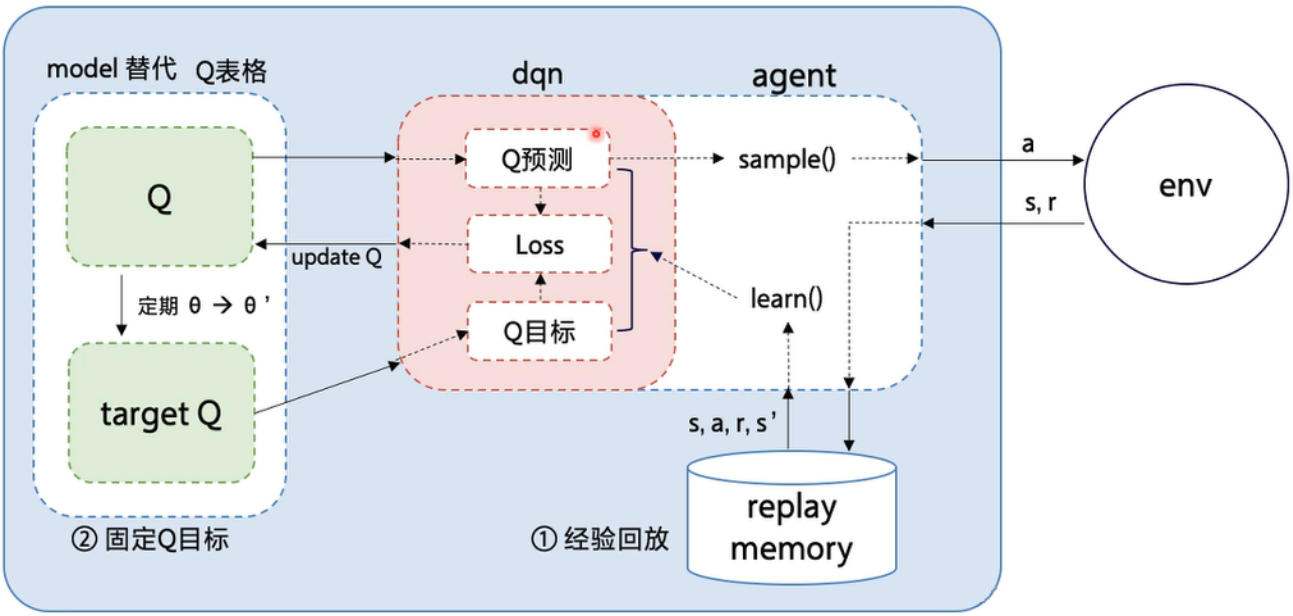
在上面的DQN中，我们可以看到在计算目标值y_j时和计算当前值用的是同一个网络Q，这样在计算目标值y_j时用到了我们需要训练的网络Q，之后我们又用目标值y_j来更新网络Q的参数，这样两者的依赖性太强，不利于算法的收敛。因此在Nature DQN中提出了使用两个网络，一个原网络Q用来选择动作，并更新参数，另一个目标网络Q'只用来计算目标值y_j，在这里目标网络Q'的参数不会进行迭代更新，而是隔一定时间从原网络Q中复制过来，因此两个网络的结构也需要完全一致，否则无法进行参数复制。　　我们简单的从公式上介绍下和DQN算法的不同，在

这里有两个网络Q和Q'，他们对应的参数分别是θ和θ'，在这里的更新频率可以自己设定，更新参数直接$\theta' = \theta$。

除此之外，我们还需要更改的就是目标值y_j的计算公式为：

$$y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma max_{a'} Q(\phi_j, a_j; \theta') & \text{for non-terminal } \phi_{j+1} \end{cases}$$

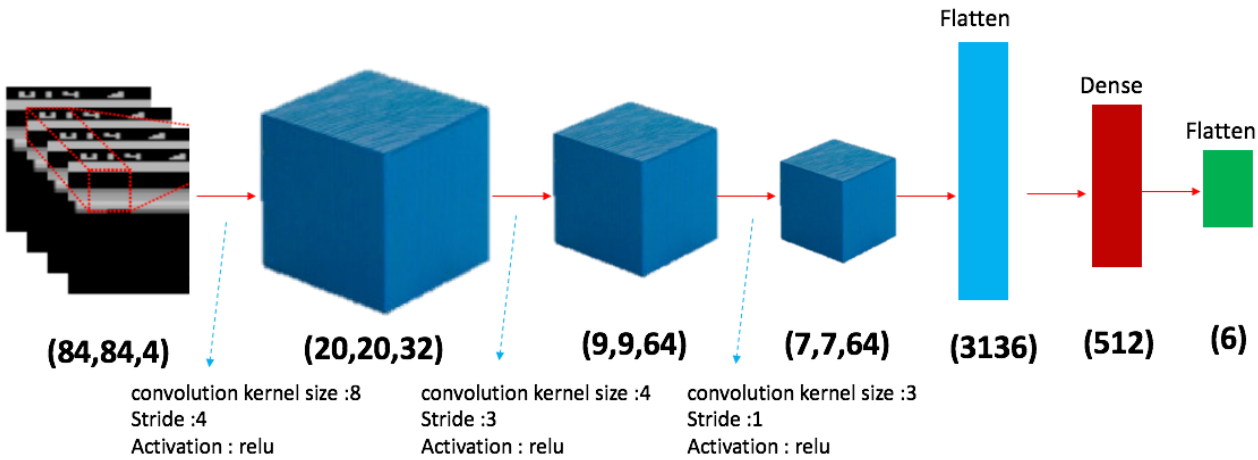
其余的和DQN完全相同。



本题的Breakout游戏的中，输入是210 * 160 * 3的图像，我们稍作处理，把边上不需要的像素去掉之后降阶采样灰度化，将84 * 84 * 4的图像作为算法的输入

神经网络的输入不是单帧的图像，而是最近的连续四帧图像作为输入。这也很好理解，因为这样就加入了时间序列。对于打砖块这个游戏，如果只用一帧作输入的话，虽然砖块在同一个位置，但是可能是向好几个方向运动的，agent无法判断它的价值。但是如果我们添加了最近几帧，agent就可以根据前后的时间判断出是向哪个方向运动的，这个状态就完整了。

下图显示了使用卷积神经网络对于图形的处理结构：



4 代码结构分析

Base implementation: https://gitee.com/goluke/dqn-breakout

4.1 `main.py`

`main.py`首先定义了需要的常量

```
GAMMA = 0.99            # discount factor
GLOBAL_SEED = 0         # global seed initialize
MEM_SIZE = 100_000      # memory size
RENDER = False          # if true, render gameplay frames
STACK_SIZE = 4          # stack size

EPS_START = 1           # starting epsilon for epsilon-greedy alogrithm
EPS_END = 0.1           # after decay steps, spsilon will reach this and keep
EPS_DECAY = 1000000     # steps for epsilon to decay

BATCH_SIZE = 32         # batch size of TD-learning traning value network
POLICY_UPDATE = 4       # policy network update frequency
TARGET_UPDATE = 10_000  # target network update frequency
WARM_STEPS = 50_000     # warming steps before training
MAX_STEPS = 50_000_000  # max training steps
EVALUATE_FREQ = 100_000 # evaluate frequency
```

- `GAMMA` 是折扣（衰减）因子 γ，设为 `0.99`；
- `MEM_SIZE` 是 `ReplayMemory` 中的 `capacity`；
- `RENDER` 为 `TRUE` 的时候在每次评价的时候都会渲染游戏画面；
- `STACK_SIZE` 是 `ReplayMemory` 中的 `channels`；
- `EPS_START` 和 `EPS_END` 是在 `EPS_DECAY` 步中 ϵ 衰减的开始和结尾值，之后 ϵ 一直保持在 `EPS_END`，值得一提的是一开始 `EPS_START` 会是 `1`，但是后面加载模型继续训练的时候有必要更改成较小的数值，否则加载的模型的性能不能很好地表现；
- `BATCH_SIZE` 是在从 `ReplayMemory` 中取样的时候的取样个数；
- `POLICY_UPDATE` 是策略网络更新的频率；
- `TARGET_UPDATE` 是目标网络更新的频率；
- `WARM_STEPS` 是为了等到 `ReplayMemory` 中有足够的记录的时候再开始降低 ϵ；
- `MAX_STEPS` 是训练的步数；
- `EVALUATE_FREQ` 是评价的频率。

接着初始化随机数、初始化计算设备、初始化环境 `MyEnv`、智能体 `Agent` 和 `ReplayMemory`。

注意此处把 `done` 置为 `True` 是为了开始训练时初始化环境并记录一开始的观察。

然后开始实现上面所说的 **Nature DQN** 算法，在循环中首先判断一个回合是否已经结束，若结束则重置环境状态，并将观察数据入队存储：

```
if done:
    observations, _, _ = env.reset()
    for obs in observations:
        obs_queue.append(obs)
```

接着判断是否已经经过 `Warming steps`，若是，则将 `training` 置为 `True`，此时则会开始衰减 ϵ：

```
training = len(memory) > WARM_STEPS
```

接着观察现在的状态 `state`，并根据状态选择动作 `action`，然后获得观察到的新的信息 `obs`、反馈 `reward` 和是否结束游戏的状态 `done`：

```
state = env.make_state(obs_queue).to(device).float()
action = agent.run(state, training)
obs, reward, done = env.step(action)
```

把观察入队，把当前状态、动作、反馈、是否结束都记录入 `MemoryReplay`：

```
obs_queue.append(obs)
memory.push(env.make_folded_state(obs_queue), action, reward, done)
```

更新策略网络和同步目标网络，同步目标网络就是把目标网络的参数更新为策略网络的参数：

```
if step % POLICY_UPDATE == 0 and training:
    agent.learn(memory, BATCH_SIZE)
if step % TARGET_UPDATE == 0:
    agent.sync()
```

评价当前网络，将平均反馈和训练出来的策略网络保存，并结束游戏。若 `RENDER` 为 `True` 则渲染游戏画面：

```
 if step % EVALUATE_FREQ == 0:
        avg_reward, frames = env.evaluate(obs_queue, agent, render=RENDER)
        with open("rewards.txt", "a") as fp:
            fp.write(f"{step//EVALUATE_FREQ:3d} {step:8d} {avg_reward:.1f}\n")
        if RENDER:
            prefix = f"eval_{step//EVALUATE_FREQ:03d}"
            os.mkdir(prefix)
            for ind, frame in enumerate(frames):
                with open(os.path.join(prefix, f"{ind:06d}.png"), "wb") as fp:
                    frame.save(fp, format="png")
        agent.save(os.path.join(
            SAVE_PREFIX, f"model_{step//EVALUATE_FREQ:03d}"))
        done = True
```

4.2 `utils_drl.py`

`utils_drl.py` 中实现了智能体 `Agent` 类，初始化了传入参数和两个模型，并且在没有传入训练好的模型的时候初始化模型参数，在传入训练好的模型的时候加载模型参数。

```
if restore is None:
            self.__policy.apply(DQN.init_weights)
        else:
            self.__policy.load_state_dict(torch.load(restore))
        self.__target.load_state_dict(self.__policy.state_dict())
        self.__optimizer = optim.Adam(
            self.__policy.parameters(),
            lr=0.0000625,
            eps=1.5e-4,
        )
        self.__target.eval()
```

`Agent` 类中定义了四个函数，分别如下：

- `run()` 函数实现了根据 ϵ-greedy 策略选择一个动作；
- `learn()` 函数实现了更新神经网络的参数的功能：

```
def learn(self, memory: ReplayMemory, batch_size: int) -> float:
    """learn trains the value network via TD-learning."""
    # 从memory中随机取样本
    state_batch, action_batch, reward_batch, next_batch, done_batch = \
            memory.sample(batch_size)
    # state预期的values
    values = self.__policy(state_batch.float()).gather(1, action_batch)

    # max_action = self.__policy(next_batch.float()).max(1)
    # values_next = self.__target(next_batch.float()).gather(1, max_action)

    # 下一个state预期的values
    values_next = self.__target(next_batch.float()).max(1).values.detach()
    # state现实的values=衰减因子gamma*values_next+reward
    expected = (self.__gamma * values_next.unsqueeze(1)) * \
            (1. - done_batch) + reward_batch
    # 损失
    loss = F.smooth_l1_loss(values, expected)

    self.__optimizer.zero_grad()
    # 求导
    loss.backward()
    for param in self.__policy.parameters():
      param.grad.data.clamp_(-1, 1)
      #更新policy神经网络的参数
    self.__optimizer.step()

    return loss.item()
```

- `sync()` 函数将目标网络延时更新为策略网络；
- `save()` 函数保存当前的策略网络参数。

4.3 `utiles_env.py`

`utils_env.py` 主要实现了调用包并配置运行的游戏环境 `MyEnv`，主要的几个函数如下：

- `reset()` 初始化游戏并提供5步的时间让智能体观察环境；
- `step()` 执行一步动作，返回最新的观察，反馈和游戏是否结束的布尔值；
- `evaluate()` 使用给定的智能体模型来运行游戏并返回平均反馈值和记录游戏的帧。

4.4 `utils_model.py`

`utils_model.py` 中使用 `pytorch` 实现了 Nature-DQN 模型。

4.5 `utils_memory.py`

`utils_memory.py` 中主要是 `ReplayMemory` 的实现，实现了数据存储和随机抽样。

`ReplayMemory`是有界大小的循环缓冲区，用于保存最近观察到的转换。 它还实现了`sample()`方法，用于选择随机的过渡批量进行训练。

5 Double DQN 提高性能

5.1 DQN的目标Q值计算问题

在DDQN之前，基本上所有的目标Q值都是通过贪婪法直接得到的，无论是Q-Learning， DQN(NIPS 2013)还是Nature DQN，都是如此。比如对于Nature DQN,虽然用了两个Q网络并使用目标Q网络计算Q值，其第j个样本的目标Q值的计算还是贪婪法得到的，计算入下式：

$$y_j = \begin{cases} R_j & \text{is\_end}_j \text{ is true} \\ R_j + \gamma max_{a'} Q'(\phi(S'_j), A'_j, w') & \text{is\_end}_j \text{ is false} \end{cases}$$

使用max虽然可以快速让Q值向可能的优化目标靠拢，但是很容易过犹不及，导致过度估计(Over Estimation)，所谓过度估计就是最终我们得到的算法模型有很大的偏差(bias)。为了解决这个问题， DDQN通过解耦目标Q值动作的选择和目标Q值的计算这两步，来达到消除过度估计的问题。

5.2 Double DQN的算法建模

DDQN和Nature DQN一样，也有一样的两个Q网络结构。在Nature DQN的基础上，通过解耦目标Q值动作的选择和目标Q值的计算这两步，来消除过度估计的问题。

在上面，Nature DQN对于非终止状态，其目标Q值的计算式子是：

$$y_j = R_j + \gamma \max_{a'} Q'(\phi(S'_j), A'_j, w')$$

在DDQN这里，不再是直接在目标Q网络里面找各个动作中最大Q值，而是先在当前Q网络中先找出最大Q值对应的动作，即

$$a^{max}(S'_j, w) = \arg\max_{a'} Q(\phi(S_j'), a, w)$$

然后利用这个选择出来的动作$a^{max}(S'_j, w)$在目标网络里面去计算目标Q值。即：

$$y_j = R_j + \gamma Q'(\phi(S'_j), a^{max}(S'_j, w), w')$$

综合起来写就是：

$$y_j = R_j + \gamma Q'(\phi(S'_j), \arg\max_{a'} Q(\phi(S'_j), a, w), w')$$

除了目标Q值的计算方式以外，DDQN算法和Nature DQN的算法流程完全相同。

5.3 DDQN代码修改

代码只有一个地方不一样，就是计算目标Q值的时候

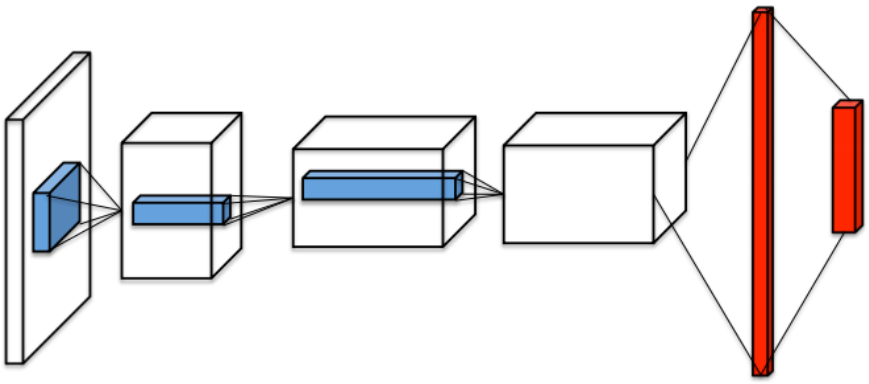
在`utils_drl.py`中对`learn()`函数进行修改

```
values = self.__policy(state_batch.float()).gather(1, action_batch)
# DDQN
max_action = self.__policy(next_batch.float()).max(1)[1]
values_next = self.__target(next_batch.float()).gather(1, max_action.unsqueeze(1)).squeeze(1)
# DQN
# values_next = self.__target(next_batch.float()).max(1).values.detach()
expected = (self.__gamma * values_next.unsqueeze(1)) * \
(1. - done_batch) + reward_batch
```

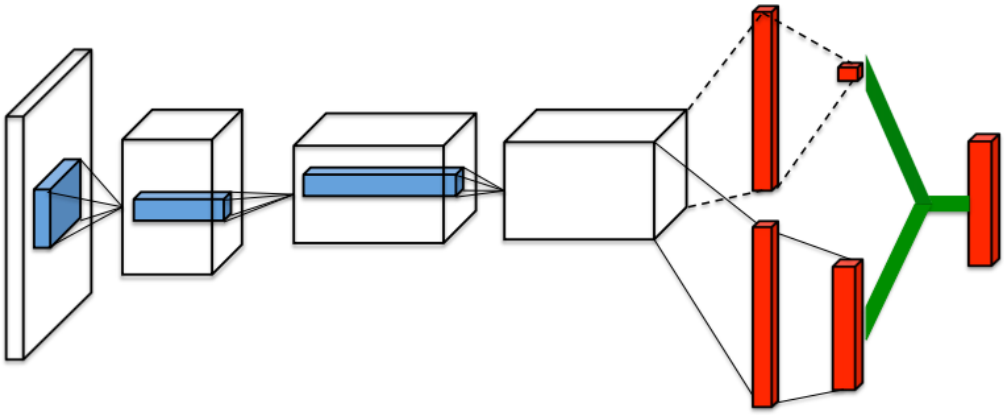
6 Dueling DQN

6.1 Dueling DQN算法建模

Dueling DQN算法提出了一种新的神经网络结构——对偶网络(duel network)。网络的输入与DQN和DDQN算法的输入一样，均为状态信息，但是输出却有所不同。Dueling DQN算法的输出包括两个分支，分别是该状态的状态价值V(标量)和每个动作的优势值A(与动作空间同维度的向量)。DQN和DDQN算法的输出只有一个分支，为该状态下每个动作的动作价值(与动作空间同维度的向量)。具体的网络结构如下图所示：



单分支网络结构(DQN和DDQN)



对偶网络结构(Dueling DQN)

在DQN算法的网络结构中，输入为一张或多张照片，利用卷积网络提取图像特征，之后经过全连接层输出每个动作的动作价值；在Dueling DQN算法的网络结构中，输入同样为一张或多张照片，然后利用卷积网络提取图像特征获取特征向量，输出时会经过两个全连接层分支，分别对应状态价值和优势值，最后将状态价值和优势值相加即可得到每个动作的动作价值(即绿色连线操作)。

Dueling DQN也是在DQN的基础上进行的改进。改动的地方是DQN神经网络的最后一层，原本的最后一层是一个全连接层，经过该层后输出n个Q值（n代表可选择的动作个数）。而Dueling DQN不直接训练得到这n个Q值，它通过训练得到的是两个间接的变量V(state value)和A(action advantage)，然后通过它们的和来表示Q值。

$$V^{\pi}(s) = E_{a\ \pi(s)}[Q^{\pi}(s,a)]$$

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$$

V代表了在当前状态s下，Q值的平均期望（综合考虑了所有可选动作）。A代表了在选择动作a时Q值超出期望值的多少。两者相加就是实际的Q(s,a)。就像当人看到某一游戏画面时是可以大概判断出当前局势的好坏，然后再根据每一种选择看哪一种action对当前局势增益最多。V就是当前局势，A就是对当前局势的增益。所以这样设计模型就是为了让神经网络对给定的s有一个基本的判断，在这个基础上再根据不同的action进行修正。

但是按照上述想法直接训练是有问题的，问题就在于若神经网络把V训练成固定值0后，就相当于普通的DQN网络了，因为此时的A值就是Q值。所以我们需要给我们的神经网络加一个约束条件，让所有动作对应的A值之和为零，使得训练出的V值是所有n个Q值的平均值（n代表可选择的动作个数）。论文中的具体做法是利用下图中的公式给到约束。

$$Q(s,a;w) = V(S;w,a) + (A(S,A;w,\beta) - \frac{1}{\mathcal{A}}\sum_{a' \in |\mathcal{A}|} A(a,a';w,\beta))$$

公式里括号中的部分其实就是之前说的A值，公式里的|A|代表了可选择动作的个数（就是上一段的n）。可以明显看出若把|A|个动作对应的括号中的部分相加，它们的和为零。所以问题就转化为利用神经网络求上述公式中的$V(s;\theta,\beta)$与$A(s,a;\theta,\alpha)$。其中$V(s;\theta,\beta)$就是前文所提到的V值，而$A(s,a;\theta,\alpha)$和前文所提到的广义上的A值其实不一样，但可以通过$A(s,a;\theta,\alpha)$计算出A值。

6.2 Dueling DQN代码实现

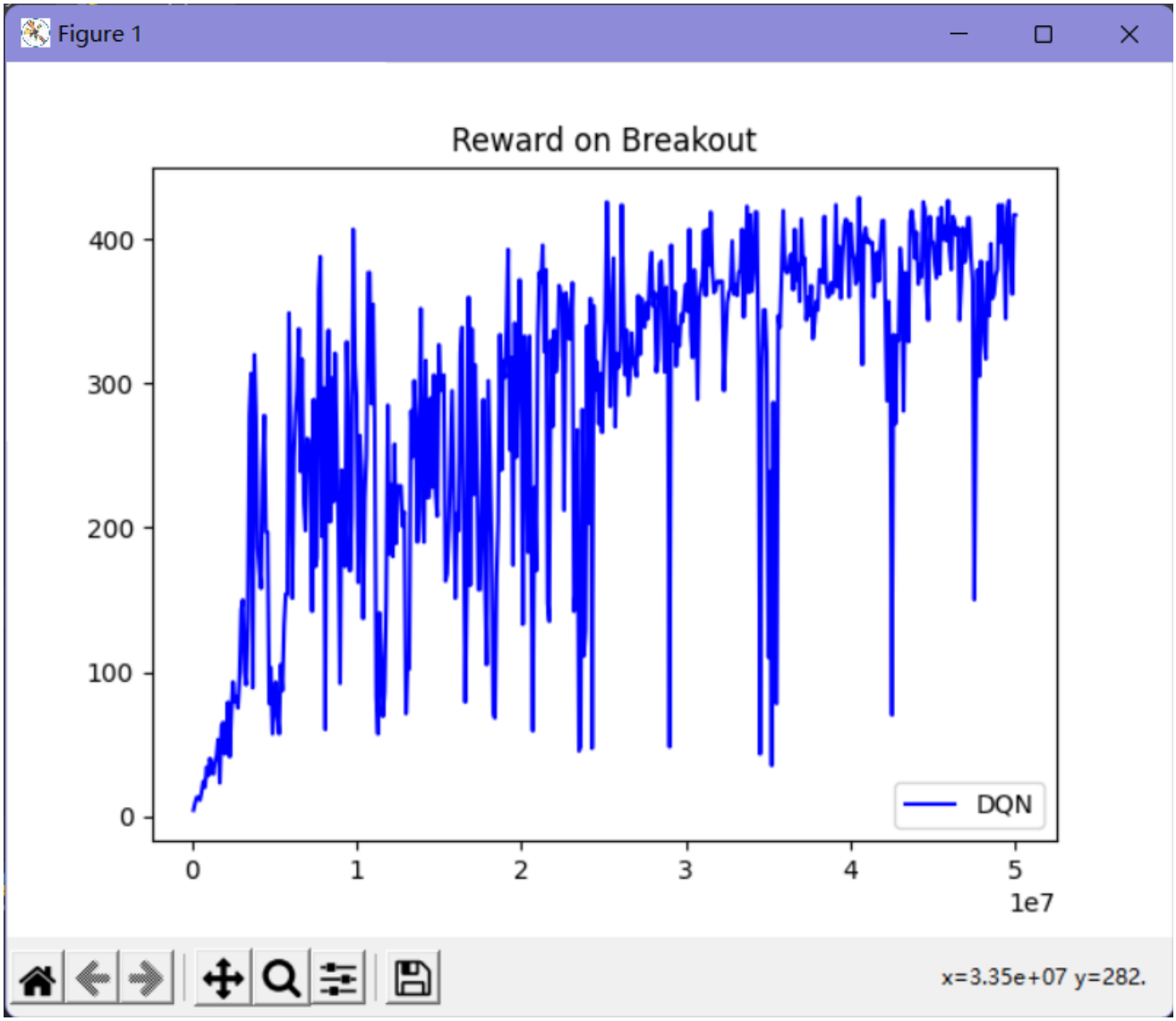
在`utils_model.py`中修改

```
class Dueling-DQN(nn.Module):
```

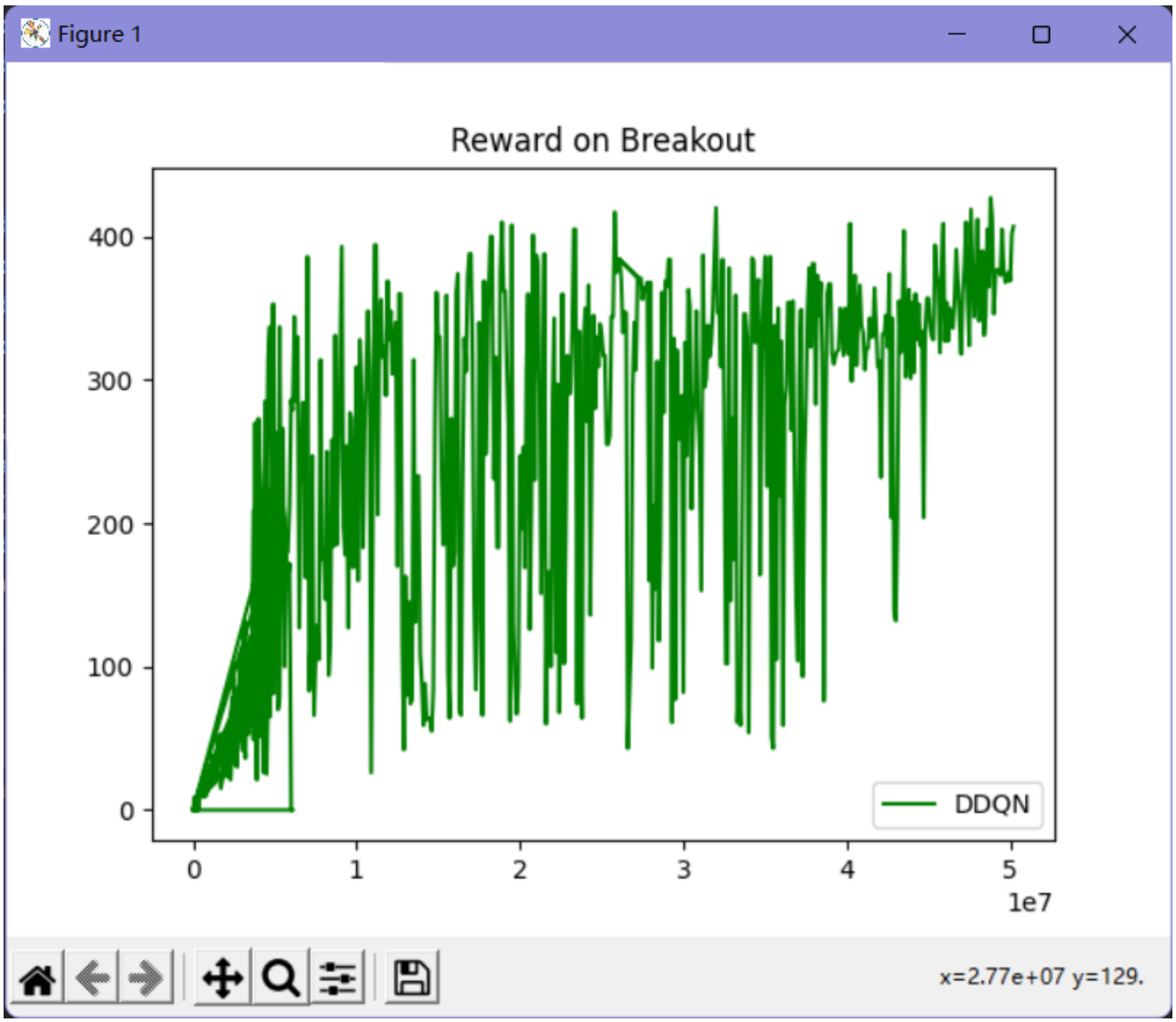
```
    def __init__(self, action_dim, device):
        super(DQN, self).__init__()
        self.__conv1 = nn.Conv2d(4, 32, kernel_size=8, stride=4, bias=False)
        self.__conv2 = nn.Conv2d(32, 64, kernel_size=4, stride=2, bias=False)
        self.__conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1, bias=False)
        self.__fc1_advantage = nn.Linear(64*7*7, 512)
        self.__fc1_value = nn.Linear(64*7*7, 512)
        self.__fc2_advantage = nn.Linear(512, action_dim)
        self.__fc2_value = nn.Linear(512, 1)
        self.__act_dim = action_dim
        self.__device = device

    def forward(self, x):
        x = x / 255.
        x = F.relu(self.__conv1(x))
        x = F.relu(self.__conv2(x))
        x = F.relu(self.__conv3(x))
        x_v = x.view(x.size(0), -1)
        v_s = self.__fc2_value(F.relu(self.__fc1_value(x_v))).expand(x.size(0), self.__act_dim)
        asa = self.__fc2_advantage(F.relu(self.__fc1_advantage(x_v)))
        return v_s + asa - asa.mean(1).unsqueeze(1).expand(x.size(0),self.__act_dim)
```

7 实验结果

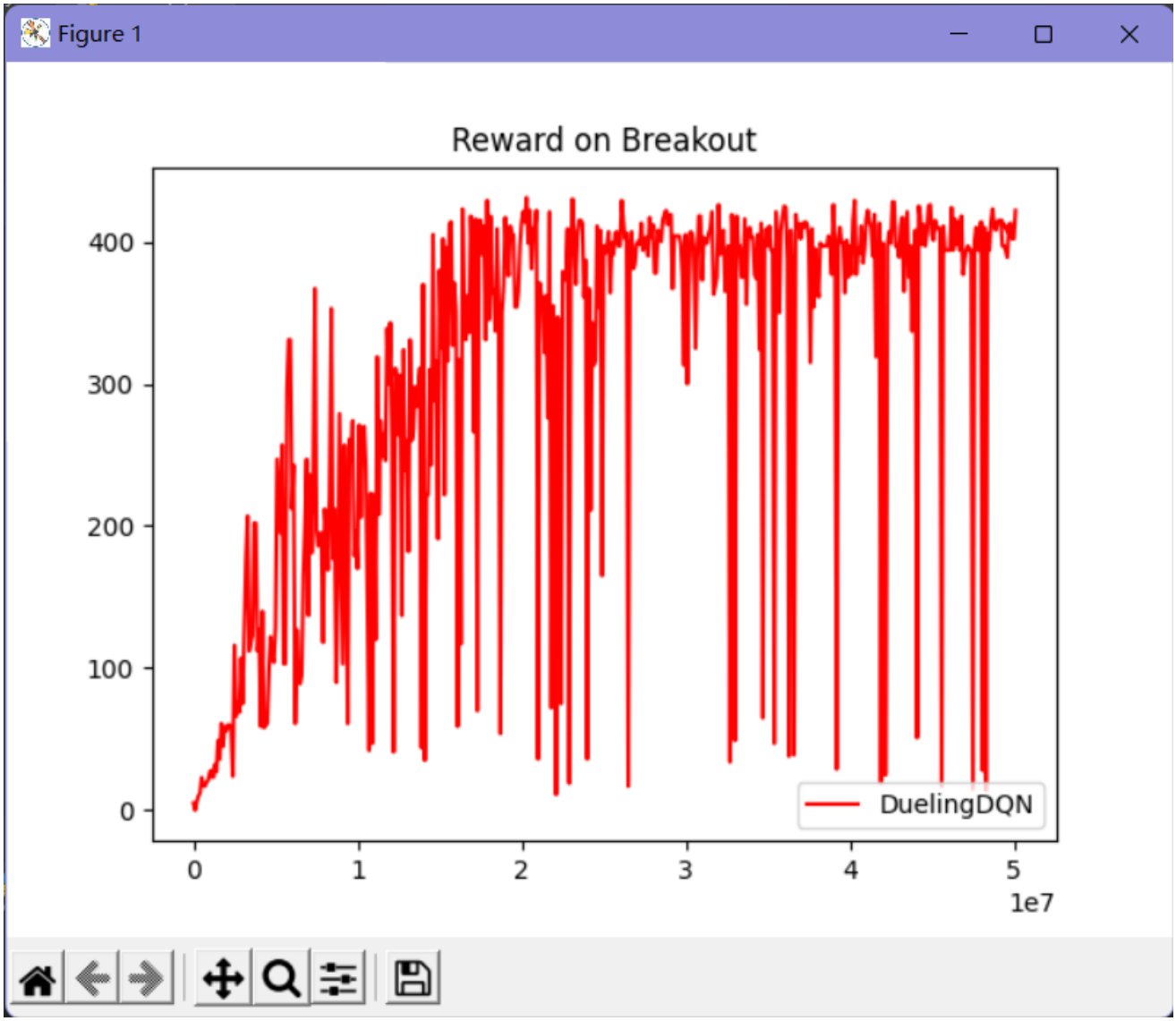
分别对三种方法进行了5000万次的迭代实验，将得出的rewards绘制成折线图



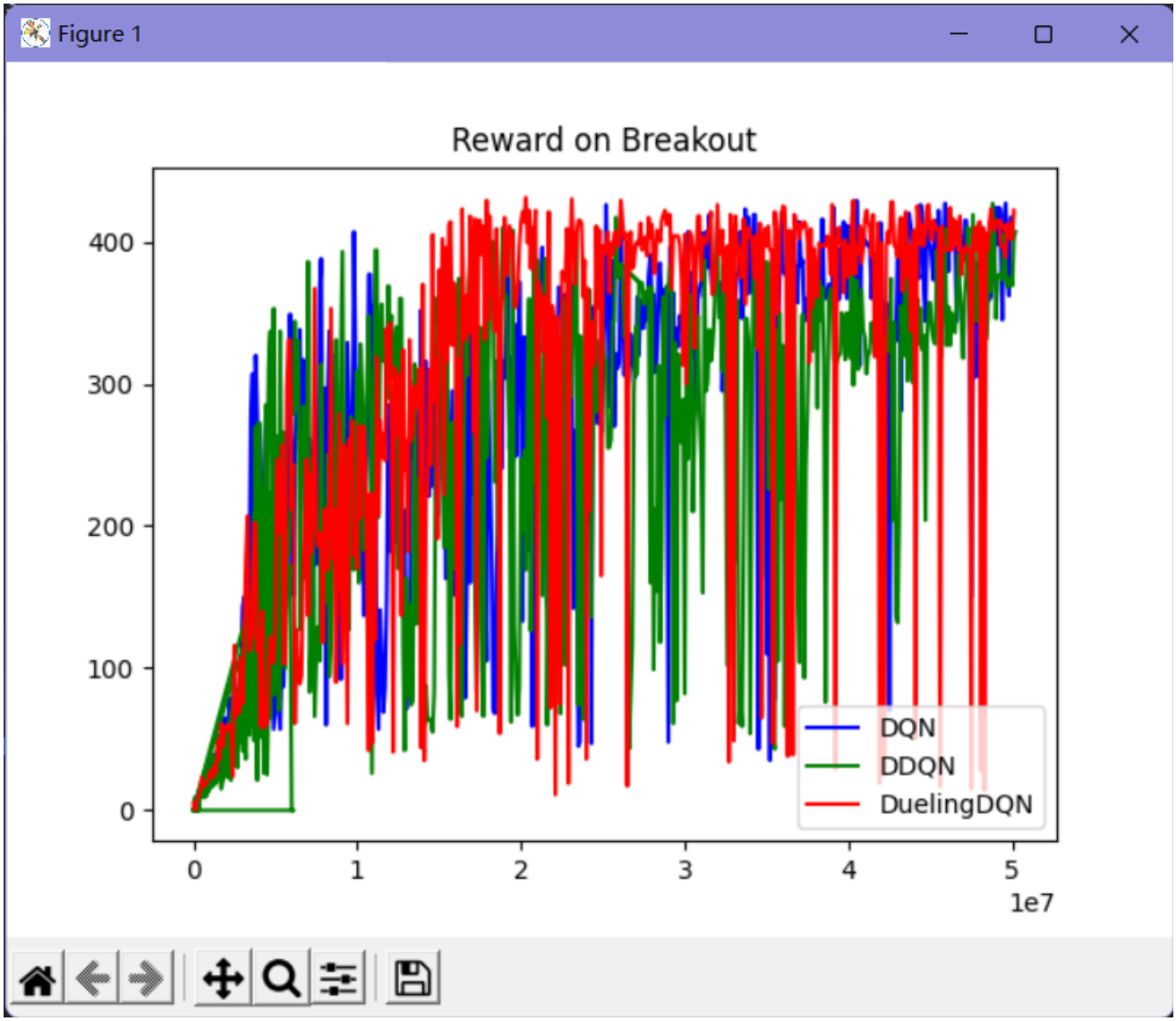
3000万次之后的数据较为稳定，相较于DQN，DDQN的更加稳定，并且接近5000万时几乎都能达到400以上



DuelingDQN在接近2000万次时就已经逐渐稳定在400上下，可以看到它明显优于前两种方法



将三种方法的结果绘制在一张图上，可以看到，Dueling-DQN的效果更好。



生成的mp4附在了代码文件夹内

8 实验分工

宋丽铭 20337234 代码修改、中文版实验报告撰写

李亚茹 20337066 搜集资料、英文版实验报告翻译

【参考资料】

Deep Learning for Video Game Playing

Playing Atari with Deep Reinforcement Learning

Human-level Control Through Deep Reinforcement Learning

Deep Reinforcement Learning with Double Q-learning

Dueling Network Architectures for Deep Reinforcement Learning