# 高级算法 期中大作业

> TRY&YMX

# 一、源代码分析

## （一）代码含义解释

### 1、main.py

主函数部分主要是**实现迭代训练**，每次迭代由环境`env`产生一个状态，基于这个状态，agent产生一个action，环境`env`基于这个action反应输出相应的reward、观察得到的observation以及是否是done的布尔量。这样每进行4个step，就会更新agent的policy网络；每进行1000个step就同步更新agent的target网络；最后每进行10000个step就评估一次模型的效果，然后重置整个游戏环境。

### 2、utils_drl.py

这个部分主要是定义了**agent类**，agent即指代游戏中的移动反弹平板。`agent`类中主要定义了它的几个行为函数，**run、learn、syn和save**。

- **run**

  run函数输入是一个状态，agent基于输入状态采用$\epsilon-greedy$方法选择action，作为输出。另外，agent会以training为判断条件，动态地对$\epsilon$进行衰减。

- **learn**

  learn函数用于训练agent的policy网络，输入是经验池和batch_size。每进行4个step，主函数就会调用agent的learn函数。它的实现是从经验池中取出batch_size个经验，基于policy网络计算出各个经验的当前$Q_j$值，然后在target网络里选择reward最大的action，计算出下一状态的$Q_{j+1}$值，然后按照公式

  $$y_j=\begin{cases}R_j & if \quad done\\ R_j+\gamma max_{a'}Q'(\phi(S'_j),A'_j,w') & if \quad not \quad done\end{cases}$$

  得到目标$Q_j$值，然后根据当前目标Q值和当前Q值之间的差距计算误差来梯度下降，然后后向传播误差更新policy网络。

- **sync**

  sync函数是为了同步target网络和policy网络，就是把policy网络的信息同步到target网络中。每进行1000个step会被主函数调用。

- **save**

  save函数保存policy网络的结果。

### 3、utils_env.py

这个部分主要是定义了游戏的**环境类MyEnv**。主要分成**reset、step、get_frame、evaluate**这几个主要函数。

- **reset**

  reset函数是重置整个游戏环境，通过初始reward为0，清空observations和frames等。

- **step**

  step是agent作出action以后，环境MyEnv针对action进行observe，输出可以得到的reward，观察到的obs和是否到达最终状态的done布尔量。

- **get_frame**

  取得帧，从而可以构建视频。

- **evaluate**

  该函数用于评估模型效果。是基于多个step得到的reward求出平均值作为平均reward，并返回截取到的帧，作为模型效果的评测指标。每进行10000个step会被主函数调用。

### 4、utils_memory.py

这个部分定义了**回放经验池的类ReplayMemory**。回放经验池的作用就是为了把过去的经验都存下来，需要训练网络时就从经验池中提取经验作为训练数据。它定义了**push、sample、len**三个函数。

- **push**

  实现把一个经验存入到回放经验池列表中，一个经验包括state，对应的action、得到的reward和done状态。

- **sample**

  实现从经验池中采样batch_size个经验用于训练agent的policy网络。输入是batch_size，然后随机选择batch_size个不同的下标，提取出经验池列表对应下标的state、next_state、action、reward和done作为输出的经验。即采取等可能的方式采样经验。

- **len**

  返回经验池大小

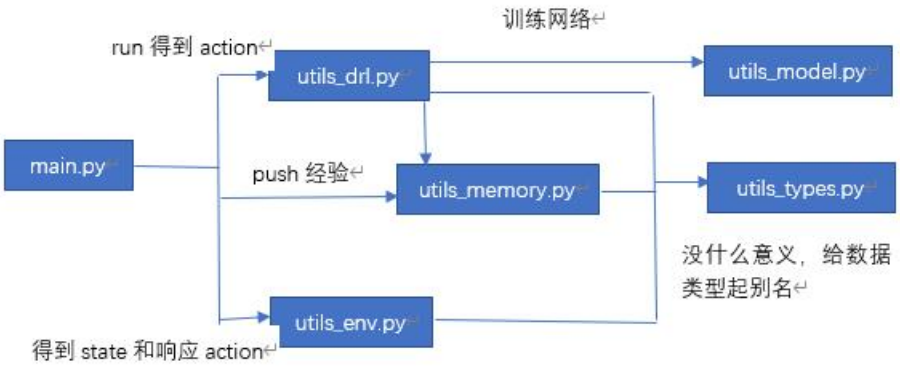
### 5、utils_model.py

这个部分**构建DQN神经网络**，实现计算Q值。采取的是CNN网络构建方法，由三个卷积层和两个全连接层构成。**forward函数**通过让输入通过神经网络，输入对应的Q值。

### 6、utils.types.py

这个部分主要是介绍各个变量，方便我们读懂代码。

## (二) 各部分的关系




# 二、优化

本次实验对优化一、优化二进行了实现。

## （一）优先经验回放（Prioritized Experience Replay）

### 1、背景

在Prioritized Experience Replay（下简称“PER”）之前，DQN和DQN变体都是通过经验回放来采样进而做目标Q值的计算的。在采样的时候，对所有的样本“一视同仁”，即在经验回放池里面的所有的样本都有相同的被采样到的概率。

然而可以发现，在经验回放池里面的不同的样本由于TD误差的不同，对反向传播的作用是不一样的。TD误差的绝对值越大，对反向传播的作用越大；反之，对反向梯度的计算影响不大。在Q网络中，TD误差 $\delta(t)$ 就是目标Q网络（target_Q_network）计算的目标Q值和当前Q网络（policy_Q_network）计算的Q值之间的差距。 因此，可以知道 $|\delta(t)|$ 越大越容易被采样，算法也越容易收敛。因此，论文中提出Prioritized Replay RQN的思路。

### 2、算法思路

- 论文中给出的**伪代码**：

**Algorithm 1** Double DQN with proportional prioritization

**Input:** minibatch $k$, step-size $\eta$, replay period $K$ and size $N$, exponents $\alpha$ and $\beta$, budget $T$.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe $S_0$ and choose $A_0 \sim \pi_\theta(S_0)$
**for** $t = 1$ **to** $T$ **do**
  Observe $S_t, R_t, \gamma_t$
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  **if** $t \equiv 0 \mod K$ **then**
    **for** $j = 1$ **to** $k$ **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    **end for**
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  **end if**
  Choose action $A_t \sim \pi_\theta(S_t)$
**end for**

- **引入优先级**

  基于前面的背景介绍，$PER$用每个经验的TD误差绝对值 $|\delta(t)|$，作为衡量该经验优先级的依据，计算公式如下：

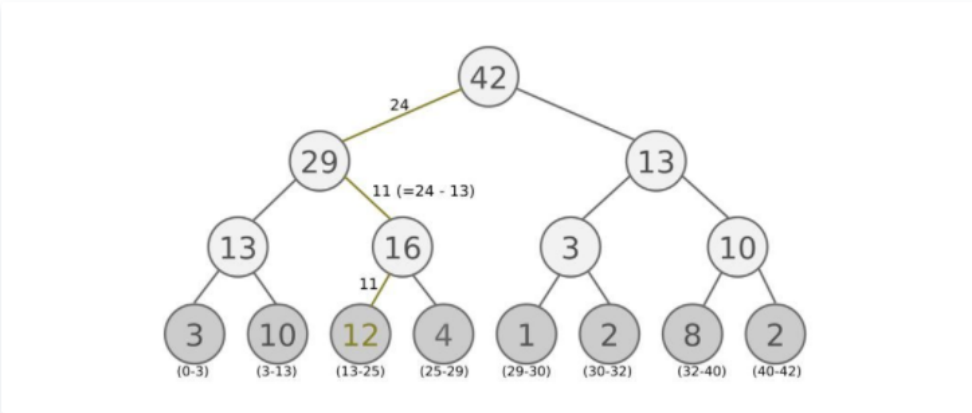
$$p_i = |\delta(t)| + \epsilon$$
$$P(i) = \frac{p_i}{\sum_j p_j}$$

  其中pi是为了防止TD error为0的经验没有被选出的概率，导致出现过拟合现象。这样一来，P(i)就是每个经验被采样的概率，TD error越大，被采样的概率就越大。这就与DQN算法的"等价采样"不同了。这也是算法的一大修改点。

- **引入sumTree**

  但是，引入优先级后，如果每次取样都要在大量经验中线性查找优先级大的经验采样，那么时间复杂度是非常大的，因此在采样时采用了sumTree的线性树结构。如下：

> **定义：** SumTree（求和树）是一种特殊的二叉树，其中父节点的值等于其子节点的值之和，如下图所示，根节点的值是所有叶子的值的和：13 = 3+10,42 = 29+13,依此类推.....




  叶节点存储的是经验优先级的大小。例如上图42是所有经验优先级的总和。例如现在要采样batch_size = 6个经验，42/6 = 7，即每个区间大小为7。这时的区间拥有的priority可能是这样.**[0-7], [7-14], [14-21], [21-28], [28-35], [35-42]**，然后在每个区间随机选取一个数(value)，比如在区间[21-28]选到了24，就按照这个24从最顶上的42开始向下搜索，首先看到最顶上的42下面有两个child nodes,拿着手中的24对比左边的child 29，如果左边的child比自己手中的值大，那就走左边这条路，接着再对比29下面左边那个点13这时手中的24比13大，那就走右边那条路，并且将手中的值根据13修改一下，变成24–13=11，接着拿着11和13左下角的12比，结果12比11大，那就选12当作这次选到的priority，并且也选择12对应的数据。之后就在这几个区间内分别做均匀采样，最后取得6个transition。

  由于要把经验和对应的优先级存入树里，所以相比原来，存储transition时还要增加往树里增加节点的步骤。

- **损失函数的优化**

$$\text{原损失函数：}\ \frac{1}{m}\sum_{j=1}^{m}(y_j - Q(S_j, A_j, w))^2$$
$$\text{考虑优先级的损失函数：}\ \frac{1}{m}\sum_{j=1}^{m} w_j(y_j - Q(S_j, A_j, w))^2$$

  其中，$w_j$是第`j`个样本的优先级权重，由TD误差$|\delta(t)|$ 归一化得到。

  - 这个优化叫做“ *Annealing the bias* ”，是为了减少bias而提出的方法。我们知道，随机更新对期望值的估计依赖于与预期相同的分布相对应的更新。而 $PER$机制引入了bias，它以一种不受控制的方式改变了这个分布，因此改变收敛结果。 然而，我们可以引入 $w_j$（也就是 importance-sample weights ）来弥补。

- **用TD误差更新优先级$p_i$，并更新到`SumTree`**

由于新产生的transition还不知道TD error，实际第一次对应的优先级大小设置为$p_i = max_{j<i}\ p_j$。那么当该transition从经验池被采样后，要用计算所得的TD error更新其pi，并且将其更新到树中。

## 3、代码

针对优化，我们在`utils.drl`和`utils.memory`中进行了修改。

- **引入`SumTree`**：在`utils.memory`中，增加了`Tree`结构及相关操作。
  - 添加了`Tree`类：
    - `add`函数：将transition和对应优先级作为新节点加入树中。
    - `updatetree`函数：因为更新了transition的优先级p，所以要用此函数更新树的节点大小。
    - `get_leaf`函数：根据输入的v，得到对应叶节点的优先级p大小和对应的transition信息。

```
class Tree(object):
    data_pointer = 0

    #存经验的下标和对应的优先级在树中
    def __init__(self,channels,capacity,device):
        self.device = device
        self.capacity = capacity  #capacity是叶节点个数
        self.tree = np.zeros(2 * capacity -1)
        #有capacity-1个父节点，capacity个子节点存优先级
        #self.data = np.zeros(capacity+1,dtype = object) #存叶节点对应的数据
data[叶子节点编号id]=data，在本实验是经验的下标
        self.__m_states =
torch.zeros((capacity,channels,84,84),dtype=torch.uint8)
        self.__m_actions = torch.zeros((capacity, 1), dtype=torch.long)
        self.__m_rewards = torch.zeros((capacity, 1), dtype=torch.int8)
        self.__m_dones = torch.zeros((capacity, 1), dtype=torch.bool)

    def add(self, p ,state,action,reward,done):
        idx = self.data_pointer + self.capacity - 1

        #self.data[self.data_pointer] = data  #增加一个记录
        self.__m_states[self.data_pointer] = state
        self.__m_actions[self.data_pointer] = action
        self.__m_rewards[self.data_pointer] = reward
        self.__m_dones[self.data_pointer] = done
        self.updatetree(idx,p)

        self.data_pointer += 1
        if self.data_pointer >= self.capacity:  #已经超出范围
            self.data_pointer = 0

    def updatetree(self,idx,p):
        change = p - self.tree[idx]  #改变位置

        self.tree[idx] = p    #将对应位置的叶节点存的值改为p
        while idx != 0:    #这样比递归更快
            idx = (idx - 1) // 2
            self.tree[idx] += change
```

```
    #v是分好第i段的均匀采样值,从树里找对应的叶子节点拿样本数据、样本叶子节点序号和样本优先级
    def get_leaf(self,v):
        parent_idx = 0
        while True:
            cl_idx = 2 * parent_idx + 1
            cr_idx = cl_idx + 1
            if cl_idx >= len(self.tree):  #没有子节点了
                leaf_idx = parent_idx
                break
            else:
                if v <= self.tree[cl_idx]: #比左孩子的值还小
                    parent_idx = cl_idx
                else:
                    v -= self.tree[cl_idx] #在右孩子继续找
                    parent_idx = cr_idx

        data_idx = leaf_idx - self.capacity + 1
        return leaf_idx, self.tree[leaf_idx], \
            self.__m_states[data_idx, :4].to(self.device).float(), \
            self.__m_states[data_idx, 1:].to(self.device).float(), \
            self.__m_actions[data_idx].to(self.device), \
            self.__m_rewards[data_idx].to(self.device).float(), \
            self.__m_dones[data_idx].to(self.device)

    @property
    def total_p(self):
        return self.tree[0] #返回根节点
```

- 在`ReplayMemory`类中添加或修改了如下操作：
  - `push`：在树里存入最新数据。

    ```
    self.tree.add(p, folded_state, action, reward, done)  #在树里存入数据
    ```

  - `sample`函数：要利用batch_size，把树中的节点按照优先级分成batch_size个区间，在每个区间中均匀采样，最终取出batch_size个经验。

    ```
    for i in range(batch_size):
                start = pri_seg * i
                end = pri_seg * (i+1)
                v = np.random.uniform(start,end) #在start和end之间均匀采样一个
                idx, p, state, next, action, reward, done = self.tree.get_leaf(v) #得到相应的优先级和下标

                idxs[i] = idx
                P = p / self.tree.total_p  #为了求w
                ISweight[i,0] = np.power(P / min_prob,-self.beta)

                b_state.append(state)
                b_next.append(next)
                b_action.append(action)
                b_reward.append(reward)
                b_done.append(done)
    ```

- update函数：用更新的优先级p更新树。

```
def update(self,indices,p):
        p += self.epsilon #变种一
        clipped_error =
np.minimum(p.cpu().data.numpy(),self.abs_err_upper)
        ps = np.power(clipped_error,self.alpha)
        for idx,valueP in zip(indices,ps):
            self.tree.updatetree(idx,valueP)
```

- **损失函数优化**：在`utils.drl`中的`learn`函数，修改了损失函数的计算方式。
  - 将从`utils.memory`中的`sample`函数得到的`weights`和原来的`loss`对应相乘，修正引入了优先级带来的bias影响。
  - 损失函数计算仍用$mse$方式（均方误差）。

```
loss = (torch.FloatTensor(is_weights).to(self.__device) *
F.mse_loss(values, expected)).mean()
```

# （二）Dueling DQN

## 1、背景

在前面讲到的PER中， 通过优化经验回放池按权重采样来优化算法。而在 Dueling DQN中，我们尝试通过**优化神经网络的结构**来优化算法。

在DQN算法中，神经网络输出的`Q`值代表动作价值，那么单纯的动作价值评估会不会不准确？我们知道，`Q(s,a)`的值既和`s`有关，又和`a`有关，但是这两种“有关”的程度不一样，或者说影响力不一样。因此，论文中提出通过优化神经网络结构，来反映两个方面的差异。

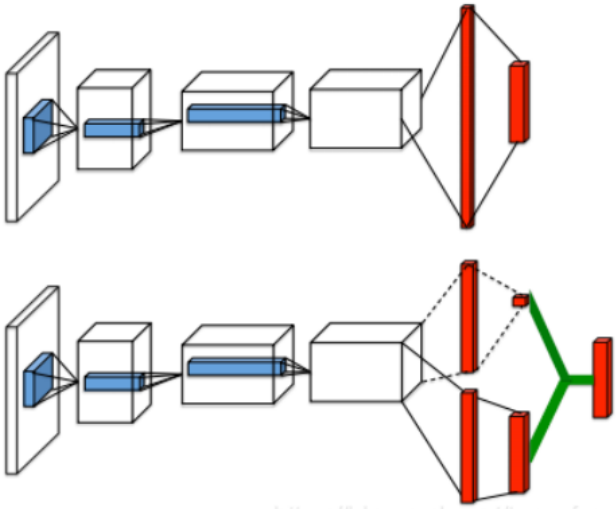
Dueling DQN考虑将Q网络分成两部分，第一部分是仅仅与状态`S`有关，与具体要采用的动作`A`无关，这部分我们叫做**价值函数部分**，记做`V(S,w,α)`。第二部分同时与状态状态`S`和动作`A`有关，这部分叫做**优势函数**(Advantage Function)部分,记为`A(S,A,w,β)`。那么最终的价值函数可以重新表示为：

$$Q(S,A,\omega,\alpha,\beta)=V(S,\omega,\alpha)+A(S,A,\omega,\beta)$$

其中，ω是公共部分的网络参数，而α是价值函数独有部分的网络参数，而β是优势函数独有部分的网络参数。

## 2、算法原理

- **网络结构图**：



- 其中，上面的是$DQN$， 其实就是CNN+全连接；
- 下面是 $Dueling\ DQN$ 的结构，只是把最后一个隐藏层拆开成两部分， 分别对应上面提到的价格函数网络部分和优势函数网络部分 ，分别输出 `V` 和 `A`，其中 `V` 只有一维，表示该状态的得分，`A` 和动作的维度是一样的，表示执行某个动作相对于该状态可以获得的额外得分。然后再由 `V` 和 `A` 通过线性组合得到 `Q` 值。
- *注意：倒数第二层隐藏层的大小和原来DQN对应的隐藏层大小是一样的，上图只不过画小了）。*

- 对优势函数做进一步的**中心化处理**，可以得到最终的线性组合公式为：

$$Q(S,A,\omega,\alpha,\beta)=V(S,\omega,\alpha)+(A(S,A,\omega,\beta)-\frac{1}{\mathcal{A}}\sum_{a'\in\mathcal{A}}A(S,a',\omega,\beta))$$

## 3、代码

本优化主要设计对`utils_model`中网络结构的修改。

- 增加一层隐藏层：
  - `fc2_adv`函数：将输入大小为512的张量缩小到大小为动作维数的张量。
  - `fc2_val`函数：将输入大小为512的张量缩小到大小为1的张量。

```
self.__fc1_adv = nn.Linear(64*7*7, 512)
self.__fc1_val = nn.Linear(64*7*7, 512)

self.__fc2_adv = nn.Linear(512, action_dim)
self.__fc2_val = nn.Linear(512, 1)
```

- 修改`forward`函数：
  - 添加`fc2`的部分，并将最后隐藏层输出的`val`和`adv`进行线性组合，输出Q值。

```
adv = F.relu(self.__fc1_adv(x.view(x.size(0), -1)))
adv = self.__fc2_adv(adv)

val = F.relu(self.__fc1_val(x.view(x.size(0), -1)))
val = self.__fc2_val(val).expand(x.size(0), self.num_actions)

res = val + adv - adv.mean(1).unsqueeze(1).expand(x.size(0),
self.num_actions)
```

# 三、结果分析

- 本次实验我们是在集群上的`cuda`环境中运行的。
- 并且，此次我们将`main.py`参数设置改成了如下数字，在迭代次数为500_000内比较实验结果：

```
TARGET_UPDATE = 10_000
WARM_STEPS = 5000
MAX_STEPS = 5_000_00
EVALUATE_FREQ = 10_000
```

- **从运行时间的角度分析：**
  - 在此情况下，$PER+Dueling\ DQN$大概需要3个小时可以跑完，而$PER\ DQN$大概需要9个小时，纯$DQN$需要10个小时+。
  - 而$PER+Dueling\ DQN$由于进行了神经网络和损失函数两方面的优化，因此速度大大提升。而由于PER DQN在损失函数方面的优化是体现在引入了`SumTree`上的，而树结构的搜索和更新都需要额外的时间和空间，所以时间和纯$DQN$的时间相比没有很明显的提升。
- **从性能提升的角度分析，实验的`reward.txt`结果如下：**

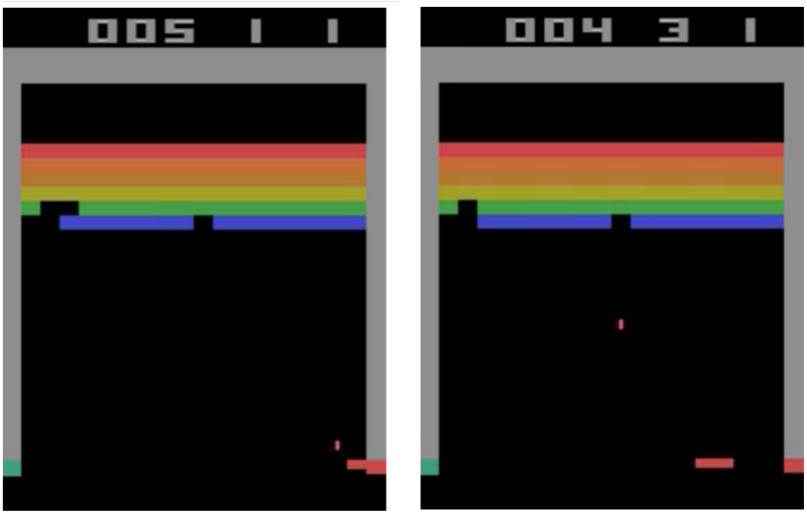
| 迭代次数 | DQN | PER DQN | PER+Dueling DQN |
|---|---|---|---|
| 0 | 4.3 | 4.3 | 4.3 |
| 100_000 | 2.7 | 1.7 | 1.7 |
| 200_000 | 1.7 | 3.0 | 4.7 |
| 300_000 | 7 | 9.0 | 8.7 |
| 400_000 | 7.3 | 8.3 | 10.7 |
| 500_000 | 8.7 | 9.3 | 12.3 |

由上面的表格可以发现，整体来说效果排名是：$PER+Dueling\ DQN > PER\ DQN > DQN$。但可以看到，并不是在所有的迭代次数中都有不等式成立。我们猜测这与500_000的迭代次数有限和实验的随机性有关，跑出来的结果都是未收敛，即欠拟合的效果。

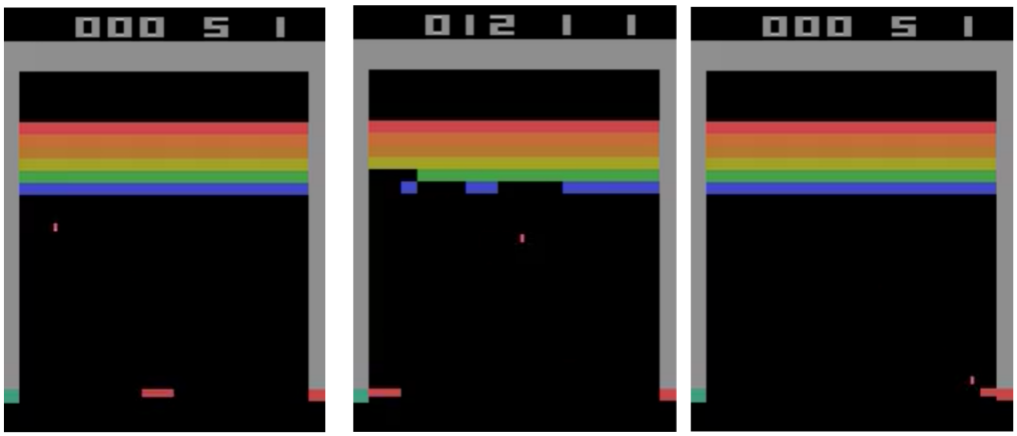
另附上$PER+Dueling\ DQN$ 的reward.txt图：

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 4.3 |
| 2 | 1 | 10000 | 1.7 |
| 3 | 0 | 0 | 4.3 |
| 4 | 0 | 0 | 4.3 |
| 5 | 1 | 10000 | 1.7 |
| 6 | 2 | 20000 | 2.7 |
| 7 | 3 | 30000 | 2.3 |
| 8 | 4 | 40000 | 3.0 |
| 9 | 5 | 50000 | 4.3 |
| 10 | 6 | 60000 | 2.0 |
| 11 | 7 | 70000 | 4.3 |
| 12 | 8 | 80000 | 4.0 |
| 13 | 9 | 90000 | 2.3 |
| 14 | 10 | 100000 | 1.7 |
| 15 | 11 | 110000 | 2.0 |
| 16 | 12 | 120000 | 1.3 |
| 17 | 13 | 130000 | 4.7 |
| 18 | 14 | 140000 | 3.0 |
| 19 | 15 | 150000 | 1.7 |
| 20 | 16 | 160000 | 4.0 |
| 21 | 17 | 170000 | 3.0 |
| 22 | 18 | 180000 | 4.0 |
| 23 | 19 | 190000 | 4.0 |
| 24 | 20 | 200000 | 3.7 |
| 25 | 21 | 210000 | 0.3 |
| 26 | 22 | 220000 | 2.7 |
| 27 | 23 | 230000 | 1.7 |
| 28 | 24 | 240000 | 5.7 |
| 29 | 25 | 250000 | 7.7 |
| 30 | 26 | 260000 | 5.7 |
| 31 | 27 | 270000 | 8.0 |
| 32 | 28 | 280000 | 8.7 |
| 33 | 29 | 290000 | 5.7 |
| 34 | 30 | 300000 | 7.3 |
| 35 | 31 | 310000 | 5.7 |
| 36 | 32 | 320000 | 6.7 |
| 37 | 33 | 330000 | 10.0 |
| 38 | 34 | 340000 | 7.7 |
| 39 | 35 | 350000 | 4.3 |
| 40 | 36 | 360000 | 9.3 |
| 41 | 37 | 370000 | 8.0 |
| 42 | 38 | 380000 | 6.3 |
| 43 | 39 | 390000 | 10.7 |
| 44 | 40 | 400000 | 5.7 |
| 45 | 41 | 410000 | 9.0 |
| 46 | 42 | 420000 | 8.7 |
| 47 | 43 | 430000 | 4.0 |
| 48 | 44 | 440000 | 9.0 |
| 49 | 45 | 450000 | 5.0 |
| 50 | 46 | 460000 | 11.3 |
| 51 | 47 | 470000 | 6.3 |
| 52 | 48 | 480000 | 5.7 |
| 53 | 49 | 490000 | 12.3 |
| 54 | 0 | 0 | 4.3 |

- **从视频的效果来看：**
  - 纯$PER$的效果：




  - $PER + Dueling\ DQN$的效果：




  - 经过比较以上的视频效果，发现纯 $PER$的paddle在相同条件下经常只会呆在右边，且抖动次数少、速度快，不会跑到中间或右边去接球，故左边和中间的经常miss。而$PER + Dueling\ DQN$的paddle在相同条件下会明显左右摆动，且抖动次数多、速度非常快，会跑到中间和左边去接球，故miss的次数减少，分数相应增加。

# 四、实验源码

详见`github`：https://github.com/SleepingMonster/DQN

# 五、组员分工

| Member | Ideas(%) | Coding(%) | Writing(%) |
| --- | --- | --- | --- |
| 唐瑞怡 | 40 | 50 | 60 |
| 叶苗欣 | 60 | 50 | 40 |

**References:**


[1] DQN系列(3): 优先级经验回放(Prioritized Experience Replay)论文阅读、原理及实现：https://cloud.tencent.com/developer/article/1633994

[2] 优先经验回放(Prioritized Experience Replay)：https://blog.csdn.net/lsjmax/article/details/102731905

[3] RL论文阅读【三】Dueling Network Architectures for Deep Reinforcement Learning：https://blog.csdn.net/taoyafan/article/details/90745419

[4] 论文笔记7：Prioritized Experience Replay：https://blog.csdn.net/yyyxxxsss/article/details/80858127

[5] 强化学习 9 —— DQN 改进算法 DDQN、Dueling DQN 详解与tensorflow 2.0实现：https://blog.csdn.net/november_chopin/article/details/107913317