# 强化学习
# 作业三

502022370062, 张含笑, zhanghx@lamda.nju.edu.cn

2022 年 12 月 15 日

## 1 作业内容

在 gym Atari 环境中实现 DQN 算法及其变体。

## 2 实现过程

在本次作业中, 我实现了 DQN 算法和 Double DQN 算法.

### 2.1 Deep Q-Learning 方法

在 Q-learning 中, 我们用一个 Qtable, 记录在每一个状态下, 各个动作的 Q 值. Qtable 的作用是当我们输入状态 S, 我们通过查表返回能够获得最大 Q 值的动作 A. 也就是我们需要找一个 S-A 的对应关系. 但在现实生活中, 很多状态并不是离散而是连续的. 在 Atari 游戏中, 状态也是连续的. DQN 算法使用神经网络代替原来的 QTable, 是一种将 Q_learning 通过神经网络近似值函数的一种方法, 函数允许连续状态的表示. 具体伪代码见算法1.

DQN 算法涉及两个重要的思想: Experience Replay 和 Target Network.

1). Experience Replay: 将系统探索环境得到的数据储存起来, 然后随机采样样本更新深度神经网络的参数. 之所以加入 experience replay, 是因为样本是从游戏中的连续帧获得的, 这与简单 RL (比如 maze) 相比样本的关联性大了很多, 如果没有 experience replay, 算法在连续一段时间内基本朝着同一个方向做梯度下降, 那么同样的步长下这样直接计算 gradient 就有可能不收敛. 因此 experience replay 是从一个 memory pool 中随机选取了一些 experience, 然后再求梯度, 从而避免了这个问题.

Experience Replay 优点:

(a) 数据利用率高，因为一个样本被多次使用.

(b) 连续样本的相关性会使参数更新的方差比较大, 该机制可减少这种相关性.

2). Target Network: 引入 Target Network 后, 在一段时间里 (`args.update_tar_interval`) 目标 Q 值是保持不变的, 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. 具体地, $Q(s, a; \theta_i)$ 表示当前网络 Main Network 的输出, 是用来评估当前状态动作

**Algorithm 1** Deep Q-Learning Algorithm

Initialize replay memory $D$ to capacity $N$

Initialize action-value function $Q$ with random weights $\theta$

Initialize target action-value function $\hat{Q}$ with random weights $\hat{\theta} = \theta$

**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 $\epsilon$ select a random action $a_t$

otherwise select $a_t = \arg\max_a Q(\phi(s_t), a; \theta)$

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 $D$

Set $Q_{Target} = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max\limits_{a'} \hat{Q}(\phi_{j+1}, a'; \hat{\theta}) & \text{otherwise} \end{cases}$

Perform a gradient descent step on $\left(Q_{Target} - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters $\theta$

Every $C$ steps reset $\hat{Q} = Q$

**end for**

**end for**

对的值函数; $\hat{Q}(s, a; \theta_i)$ 表示 Target Network 的输出, 代入算法中求 TargetQ 值的公式中得到目标 Q 值. 根据算法中定义的 Loss Function 更新 Main Network 的参数, 每经过 (`args.update_tar_interval`) 轮迭代, 将 Main Network 的参数复制给 TargetNet.

## 2.2 Vanilla DQN 方法的具体实现

### 2.2.1 定义超参数

一些较为重要的, 直接影响训练效果的超参数定义如下:

| | |
|---|---|
| `args.gamma` | 计算 targetQ 值时的折扣系数 |
| `args.epsilon` | 选择随机动作的概率, 随训练轮数衰减 |
| `args.epsilon_min` | 最终选择随机动作的概率 |
| `args.eps_decay` | 用于控制上述概率衰减的速率 |
| `args.frames` | 总训练帧数 |
| `args.learning_interval` | 每次更新参数的间隔帧数 |
| `args.update_tar_interval` | 每次更新目标网络的间隔帧数 |
| `args.batch_size` | 每次迭代抽取的样本数量 |
| `args.win_reward` | 获胜时获得的奖励 |
| `args.win_break` | 是否在获胜时退出 |

### 2.2.2 定义 CnnDDQNAgent

**__init__(self, config):** 初始化 Agent:

1. 定义 buffer 用于存储已经做过的动作和对应的状态变化, 用于随机采样以更新模型参数
2. 定义动作选择模型 (CNNDQN)
3. 初始化目标网络 Target Network 使其和主网络拥有同样的参数
4. 定义优化器

**act(self, state, epsilon):** 动作选择函数, `epsilon` 为做出随机动作的概率. 若生成的随机数大于这个概率, 则根据模型得到的 Q 值选择动作. 在训练过程中, `epsilon` 会随着游戏进行的帧数衰减, 从初始值 $\epsilon_{start}$ 衰减至 $\epsilon_{final}$, 衰减速率由 $\epsilon_{decay}$ 控制, 具体表示为

$$\epsilon = \epsilon_{final} + (\epsilon_{start} - \epsilon_{final}) * \exp(-\frac{\texttt{frame}}{\epsilon_{decay}})$$

其中 `frame` 是当前帧数, $\epsilon_{start}$ = `args.epsilon`, $\epsilon_{final}$ = `args.epsilon_min`

**learning(self, fr):** 模型学习函数, 用于计算损失并更新模型参数. 首先在 buffer 中进行采样, 得到的样本包括 [环境 `s0`, 下一时刻的环境 `s1`, 动作 `a`, reward `r`, mask `done` (用于表示正常结束或是因时间限制而结束)]. 将 `s0` 送入模型得到主模型的 Q 值, 再将 `s1` 送入 Target Network 得到目标网络的 Q 值, 根据算法中的公式

$$Q_{Target} = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max\limits_{a'} \hat{Q}(\phi_{j+1}, a'; \hat{\theta}) & \text{otherwise} \end{cases}$$

得到 TargetQ 值, 用 `done` 作为选择 TargetQ 计算方法的指示器.

计算 Q 值和 loss 的具体实现如下:

```
s0, s1, a, r, done = self.buffer.sample(self.config.batch_size)
if self.config.use_cuda:
    s0 = s0.float().to(self.config.device)/255.0
    s1 = s1.float().to(self.config.device)/255.0
    a = a.to(self.config.device)
    r = r.to(self.config.device)
    done = done.to(self.config.device)
else:
    s0 = s0.float()/255.0
    s1 = s1.float()/255.0

q_values = self.model(s0)

with torch.no_grad():
    target_q_values = self.target_model(s1)
    max_target_q_value = target_q_values.max(1)[0].unsqueeze(-1)

target_q = r + self.config.gamma * (1-done)* max_target_q_value
```

在对模型进行训练的过程中, 发现使用默认的超参数无法得到较好模型, 于是对训练流程做出了以下修改:

1. 将 `args.epsilon_min` 设置为 0.05.
2. 将总训练帧数 `args.frames` 降低为 2000000.
3. 将学习率 `args.learning_rate` 增大至 1e-4.
4. 将每批次采样数量 `args.batch_size` 增大至 128.
5. 将优化器改为 Adam.

### 2.3 DQN 变体: Double DQN 的实现

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 还是 Vanilla DQN. 因此 Vanilla DQN 有一个显著的问题, 就是 DQN 估计的 Q 值往往会偏大. 这是由于主网络 Q 值是依据下一个 state 的 Q 值的最大值来估算的, 但下一个 state 的 Q 值也是一个估算值, 也依赖它的下一个 state 的 Q 值..., 这就导致了 Q 值往往会有偏大的的情况出现.

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.

Double DQN 使用两个网络来计算 TargetQ 值, Q1 网络推荐能够获得最大 Q 值的动作, Q2 网络计算这个动作在 Q2 网络中的 Q 值. 我们可以复用 DQN 的代码, 使用 Main Network 推荐能够获得最大 Q 值的动作, 使用 Target Network 计算这个动作在目标网络中的 Q 值.

计算 Q 值和 loss 的具体实现如下:

```
s0, s1, a, r, done = self.buffer.sample(self.config.batch_size)
if self.config.use_cuda:
    s0 = s0.float().to(self.config.device)/255.0
    s1 = s1.float().to(self.config.device)/255.0
    a = a.to(self.config.device)
    r = r.to(self.config.device)
    done = done.to(self.config.device)
else:
    s0 = s0.float()/255.0
    s1 = s1.float()/255.0

q_values = self.model(s0)

with torch.no_grad():
    target_q_values = self.target_model(s1)
    extra_q_values = self.model(s1)
    actions = extra_q_values.max(1)[1]
    max_target_q_value = torch.gather(target_q_values, 1, actions.unsqueeze(1))

target_q = r + self.config.gamma * (1-done)* max_target_q_value
```

### 2.4 其他 DQN 变体

由于时间限制, 没有能够完成其他变体, 因此整理了其他变体的实现方式和个人对这些方法的理解. 个人认为 DQN with Prioritized Replay Buffer 会有比 Vanilla DQN 以及 Double DQN 更快的收敛速度, 在之后有机会会自己实现并对不同的算法的性能进行进一步的比较.

#### 2.4.1 DQN with Prioritized Replay Buffer

经验回放 (Experience replay) 让在线强化学习代理记住和重复使用过去的经验. 在 Vanilla DQN 和 Double DQN 中, experience replay 是均匀随机取样. 然而, 这种方法不考虑样本的重要性, 使得训练过程的收敛较慢. Prioritized DQN 使用了一个优先级经验的框架, 对样本的重要性进行建模作为样本的优先级, 在经验回放时更频繁地使用重要性高的样本, 从而更有效地学习.

DQN with Prioritized Replay Buffer 根据每个样本的 `TD-error` 的绝对值, 给定该样本的优先级正比于这个误差, 将这个优先级的值存入经验回放池.

由于引入了经验回放的优先级, DQN with Prioritized Replay Buffer 的经验回放池和之前的其他 DQN 算法的经验回放池就不一样了, 因为这个优先级大小会影响各个样本被采样的概率. 在实际使用中, 通常使用 SumTree 这样的二叉树结构来存储所有样本. DQN with Prioritized Replay Buffer 和 Vanilla DQN 以及 Double DQN 相比, 避免了一些没有价值的迭代, 因此收敛速度应该会有很大的提高, 是一个不错的优化点.

#### 2.4.2 Dueling DQN

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, 这部分叫做价值函数部分, 记做 $Q(S, w, \alpha)$, 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 $A(S, A, w, \beta)$. 最终的价值函数重新表示为:

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

其中, $w$ 是公共部分的网络参数, $\alpha$ 是价值函数独有部分的网络参数, 而 $\beta$ 是优势函数独有部分的网络参数.

在实际的实现中, 需要增加两个子网络结构, 分别对应上面上到价格函数网络部分和优势函数网络部分.

## 3 复现方式

在主文件夹下运行 `python code/atari_ddqn.py --train` 即可运行 DQN 算法.

若要运行基于 Double DQN 算法, 在主文件夹下运行 `python code/atari_ddqn.py --train --double-dqn True` 即可.

# 4 实验效果

累计奖励和样本训练量之间的关系如图1所示. 可以看出, 超参数相同时, Double DQN 算法和 Vanilla DQN 样本效率很接近, 即 reward 上升的速度接近; 同时两者相比小 batch size, l1-smooth loss 以及学习率为 3e-4 这几种设置都能取得更好的样本效率.

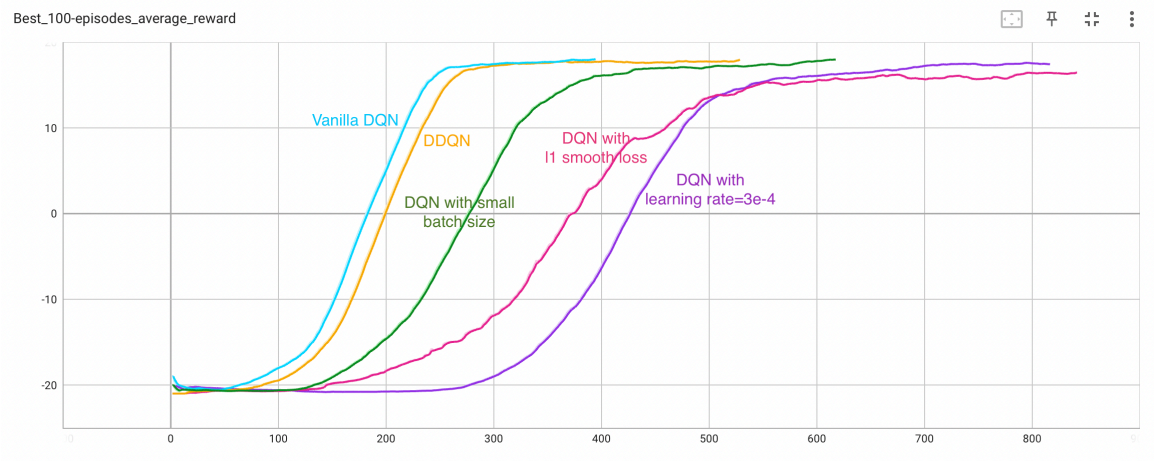


图 1: 不同算法中累计奖励和训练帧数之间的关系

# 5 小结

在这次实验中, 我发现在简单的 Atari 环境中, 基本的 CNN-based DQN 算法就能取得不错的性能; DQN 基础上的变体能进一步提升样本效率. 在超参数中, batch size 和 learning rate 较为关键, batch size 最好不要低于 128, 否则会影响样本效率; 且学习率不能设置过大, 否则收敛会很慢甚至不收敛; 选用合适的损失函数和优化器对训练结果也较为关键.

另外, 由于时间限制, 没有能够完成其他变体, 因此整理了其他变体的实现方式和个人对这些方法的理解. 个人认为 DQN with Prioritized Replay Buffer 会有比 Vanilla DQN 以及 Double DQN 更快的收敛速度, 在之后有机会会自己实现并对不同的算法的性能进行进一步的比较.