作业

Homework3

庄镇华 502022370071

A RL Homework Assignment

2022 年 12 月 14 日

问题 1

本次作业的环境为 gym 上的 Atari Game，默认为 Pong。
玩家得到的观测：一个三维数组 (12,84,84)，表示 4 帧彩色图像 (3,84,84) 的复合。
可执行的动作：离散的动作 0,1,...,18，具体数目参考具体的游戏。
奖励：玩家在游戏中每次移动都会得到奖励（可能为 0 或负数）。
游戏目标：尽可能达到高的累计奖励。
请完成：
依据 Deep Q-learning 算法，实现 DQN 及其各种变体（包括且不限于 Double DQN, DQN with Prioritized Replay Buffer, Dueling DQN 等等，**至少需要实现一种变体**），学习一个游戏策略。
绘制你实现的 Q-learning 算法的性能图（训练所用的样本与得到的累计奖励的关系图，代码中提供了 tensor board 接口，可以直接调用）。

解答：

1 作业内容

在 gym Atari 环境中实现 DQN 算法及变体（包括且不限于 Double DQN, DQN with Prioritized Replay Buffer, Dueling DQN 等，**至少需要实现一种变体**），学习一个游戏策略。

2 算法原理

2.1 DQN

算法 1 DQN 算法

初始化函数 Q、目标函数 $\hat{Q}$，令 $\hat{Q} = Q$
对于每一个回合
  对于每一个时间步
    对于给定的状态 s_t，基于 Q（$\epsilon-$ 贪婪）执行动作 a_t
    获得反馈 r，并获得新的状态 s_{t+1}
    将 (s_t, a_t, r_t, s_{t+1}) 存储到缓冲区中
    从缓冲区中采样 (通常以批量形式) (s_i, a_i, r_i, s_{i+1})
    目标值是 $y = r_i + \max Q(s_{i+1}, a)$
    更新 Q 的参数使得 $Q(s_i, a_i)$ 尽可能接近于 y (回归)
    每 C 次更新重置 $\hat{Q} = Q$

整体来说，DQN 与 Q-learning 算法的目标价值以及价值的更新方式都非常相似。主要的不同点在于：DQN 将 Q-learning 与深度学习结合，用深度网络来近似动作价值函数，而 Q-learning 则是采用表格存储；DQN 采用了经验回放的训练方法，从历史数据中随机采样，而 Q-learning 直接采用下一个状态的数据进行学习。

2.2 Double DQN

DDQN 针对 Q 值被高估的问题做出了相应改进。在 DDQN 里面，选动作的 Q 函数与计算值的 Q 函数不是同一个。即使用会更新参数的 Q 网络去选动作，而用目标 Q 网络（固定住的网络）计算值。其相对于 DQN 算法的改进主要在 Q 值计算的步骤，即由

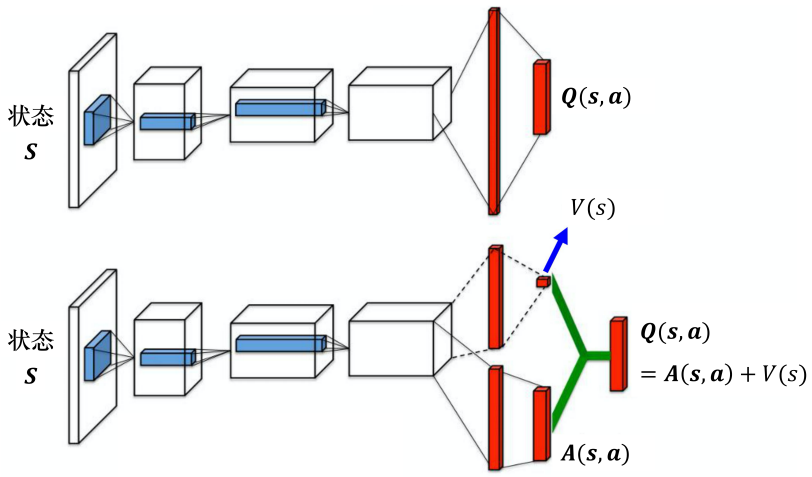
$$Q(s_t, a_t) = r_t + \max Q(s_{t+1}, a)$$

改进为

$$Q(s_t, a_t) = r_t + Q'(s_{t+1}, \text{argmax } Q(s_{t+1}, a))$$

2.3 Dueling DQN

Dueling DQN 相较于原来的 DQN 算法，唯一差别是改变了网络的架构。DQN 直接输出 Q 值，而 Dueling DQN 将 Q 值分解为状态价值 V(s) 和动作优势值 A(s, a)，通过对动作优势值 A(s, a) 施加约束，使网络倾向于改变状态价值 V(s)，而单个状态价值 V(s) 的修改会影响到所有和这个状态相关的 Q 值，即提高了经验样本的使用效率。



2.4 DQN with Prioritized Replay Buffer

DQN with Prioritized Replay Buffer 即使用了优先级经验回放的 DQN 方法。DQN 在采样数据训练 Q 网络的时候，会均匀地从回放缓冲区里面采样数据。这样不一定是最好的，因为时序差分误差较大的数据比较重要，这些数据是比较不好训练的，优先级经验回放的做法是提高这些样本的优先权，增大其被采样的概率，以实现模型的更好训练。

常用的优先级设定为，其中 p_i 指经验样本 i 的时序差分误差。

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

由于改变了样本分布，因此在计算损失的时候，需要进行重要性采样，其中 β 为需要调节的超参数。

$$w_i = (\frac{1}{N}\frac{1}{P(i)})^\beta$$

3 实现过程

3.1 实验过程

本次实验代码上实现了 DQN 算法及其三种变体（包括 Double DQN, DQN with Prioritized Replay Buffer 和 Dueling DQN）。由于时间关系以及超参数需要调节，实验效果上，将 DQN 算法和 Double DQN 算法调节到了不错的效果。

3.1.1 DQN 相关代码与解释

loss 的计算公式如下，其具体实现如下代码所示：

$$Q(s_t, a_t) = r_t + \max Q(s_{t+1}, a)$$

```
        # Tips: function torch.gather may be helpful
        # You need to design how to calculate the loss
        loss = 0
        q_eval = self.model(s0).gather(1, a)
        q_next = self.model(s1)
        q_target = r + self.config.gamma * q_next.max(1)[0].view(-1, 1) * (1 -
    done)
        loss_fn = torch.nn.MSELoss()
        assert q_eval.shape == q_target.shape
        loss = loss_fn(q_eval, q_target)
```

3.1.2 Double DQN 相关代码与解释

loss 的计算公式如下，其具体实现如下代码所示：

$$Q(s_t, a_t) = r_t + Q'(s_{t+1}, \text{argmax } Q(s_{t+1}, a))$$

```
        # Tips: function torch.gather may be helpful
        # You need to design how to calculate the loss
        # @zhuangzh
        q_eval = self.model(s0).gather(1, a)
        q_eval_next = self.model(s1)
        max_action = q_eval_next.max(1)[1].unsqueeze(1)
        q_next = self.target_model(s1).gather(1, max_action)
        q_target = r + self.config.gamma * q_next * (1 - done)
        loss_fn = torch.nn.MSELoss()
        assert q_eval.shape == q_target.shape
        loss = loss_fn(q_eval, q_target)
```

3.1.3 Dueling DQN 相关代码与解释

Dueling DQN 网络架构实现细节如下代码所示，其中主要的改动在于将 Q 值分解为基于状态动作对的 advantage 值和基于状态的 value 值。

```
class CnnDuelingDQN(nn.Module):
    def __init__(self, inputs_shape, num_actions):
```

```
        super(CnnDuelingDQN, self).__init__()

        self.inut_shape = inputs_shape
        self.num_actions = num_actions

        self.features = nn.Sequential(
            nn.Conv2d(inputs_shape[0], 32, kernel_size=8, stride=4),
            nn.LeakyReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.LeakyReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.LeakyReLU()
        )

        self.advantage = nn.Sequential(
            nn.Linear(self.features_size(), 512),
            nn.ReLU(),
            nn.Linear(512, self.num_actions)
        )

        self.value = nn.Sequential(
            nn.Linear(self.features_size(), 512),
            nn.ReLU(),
            nn.Linear(512, 1)
        )

    def forward(self, x):
        batch_size = x.size(0)
        x = self.features(x)
        x = x.reshape(batch_size, -1)
        advantage = self.advantage(x)
        value = self.value(x)
        return value + advantage - advantage.mean()

    def features_size(self):
        return self.features(torch.zeros(1, *self.inut_shape)).view(1, -1).size(1)
```

3.1.4 DQN with Pioritized Replay Buffer 相关代码与解释

SumTree 实现

SumTree 是一种树形结构，其可以实现用堆的形式存储具有优先级别的 buffer。SumTree 的叶子节点存储每个样本的优先级 p，每个非叶子节点只有两个子节点，节点的值是两个子节点的和，所以 SumTree 的根节点就是所有优先级 p 的和。

抽取一个样本的实现方式如下，从 0 ~ p_{total} 均匀抽样一个数据，假设为 q：

1. 将根结点作为父结点，然后遍历子节点；

2. 如果左子节点的优先值比 q 大，就将左子节点作为新的父节点，然后遍历子节点；

3. 如果左子节点的优先值比 q 小，就减去左子节点的优先值，并将右子节点作为新的父节点，然后遍历子节点；

4. 直到遍历的叶子节点的优先值就是优先级时停止，根据该叶子节点对应的下标就可以从经验回放池中找到对应的经验样本。

SumTree 不仅降低了时间复杂度，即每次抽样的时间复杂度为 $\mathcal{O}(n \log n)$，而且只需要均匀抽样数据，然后依据数据结构选取对应的经验样本，大大简化了抽样过程，提高了强化学习经验样本抽取与学习的效率。

```python
class SumTree(object):
    ''' a binary tree data structure where the parent's value is the sum of its children '''
    def __init__(self, capacity):
        self.capacity = capacity
        self.tree = np.zeros(2 * capacity - 1)

    def _propagate(self, idx, change):
        parent = (idx - 1) // 2
        print(parent)
        self.tree[parent] += change
        if parent != 0:
            self._propagate(parent, change)

    def _retrieve(self, idx, s):
        left = 2 * idx + 1
        right = left + 1
        if left >= len(self.tree):
            return idx
        if s <= self.tree[left]:
            return self._retrieve(left, s)
        else:
            return self._retrieve(right, s - self.tree[left])

    def total(self):
        return self.tree[0]

    def add(self, step, pri):
        idx = step + self.capacity - 1
        self.update(idx, pri)

    def update(self, idx, pri):
        change = pri - self.tree[idx]
        self.tree[idx] = pri
        self._propagate(idx, change)

    def sample(self, s):
        idx = self._retrieve(0, s)
        return idx - self.capacity + 1
```

优先级缓存池实现

优先级缓存池和普通缓存池类似，其优先级由 SumTree 来存储，不同的地方在于每次模型更新时，计算出时序差分误差，此时也要利用误差信息来更新 SumTree 中不同经验样本的优先级，优先级计算公式为

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

并且，模型计算 loss 时，由于改变了样本分布，需要进行重要性采样，因此计算 loss 时也

需要利用样本的优先级信息，其权重计算方式为

$$w_i = (\frac{1}{N}\frac{1}{P(i)})^{\beta}$$

```python
class PriorityRolloutStorage(RolloutStorage):
    def __init__(self, config):
        self.max_buff = 2 ** (int(np.log2(config.max_buff)) + 1)
        print((int(np.log2(config.max_buff)) + 1))
        self.obs = torch.zeros([self.max_buff,  *config.state_shape], dtype=torch.uint8)
        self.next_obs = torch.zeros([self.max_buff,  *config.state_shape], dtype=torch.uint8)
        self.rewards = torch.zeros([self.max_buff,  1])
        self.actions = torch.zeros([self.max_buff, 1])
        self.actions = self.actions.long()
        self.masks = torch.ones([self.max_buff,  1])
        self.num_steps = self.max_buff
        self.step = 0
        self.current_size = 0
        self.tree = SumTree(self.max_buff)
        self.pri_max = 0.1
        self.alpha = 0.6
        self.epsilon = 0.01

    def add(self, obs, actions, rewards, next_obs, masks):
        self.obs[self.step].copy_(torch.tensor(obs[None,:], dtype=torch.uint8).squeeze(0).squeeze(0))
        self.next_obs[self.step].copy_(torch.tensor(next_obs[None,:], dtype=torch.uint8).squeeze(0).squeeze(0))
        self.actions[self.step].copy_(torch.tensor(actions, dtype=torch.float))
        self.rewards[self.step].copy_(torch.tensor(rewards, dtype=torch.float))
        self.masks[self.step].copy_(torch.tensor(masks, dtype=torch.float))
        pri = (np.abs(self.pri_max) + self.epsilon) ** self.alpha
        self.tree.add(self.step, pri)
        self.step = (self.step + 1) % self.num_steps
        self.current_size = min(self.current_size + 1, self.num_steps)


    def sample(self, batch_size):
        indices = []
        segment = self.tree.total() / batch_size

        for i in range(batch_size):
            s = random.uniform(segment * i, segment * (i + 1))
            idx = self.tree.sample(s)
            indices.append(idx)

        obs_batch = self.obs[indices]
        obs_next_batch = self.next_obs[indices]
        actions_batch = self.actions[indices]
        rewards_batch = self.rewards[indices]
        masks_batch = self.masks[indices]
        return indices, obs_batch, obs_next_batch, actions_batch, rewards_batch,
```

```
    masks_batch

46
47  def update(self, idxs, td_errs):
48      self.pri_max = max(self.pri_max, max(np.abs(td_errs)))
49      for i, idx in enumerate(idxs):
50          pri = (np.abs(td_errs[i]) + self.epsilon) ** self.alpha
51          self.tree.update(idx, pri)
```

更新样本优先级

```
1       td_errs = (q_eval - q_target).cpu().squeeze().tolist()
2       self.buffer.update(idxs, td_errs)
```

3.2 相关改进

- 学习率由 $1e-6$ 调整为 $5e-5$
- 优化器由 *RMSprop* 调整为 *Adam*
- 为了消除梯度噪声，进行梯度裁剪 `param.grad.data.clamp_(-50, 50)`.

4 复现方式

复现 DQN 在主文件夹下运行 `python atari_dqn.py --train`.

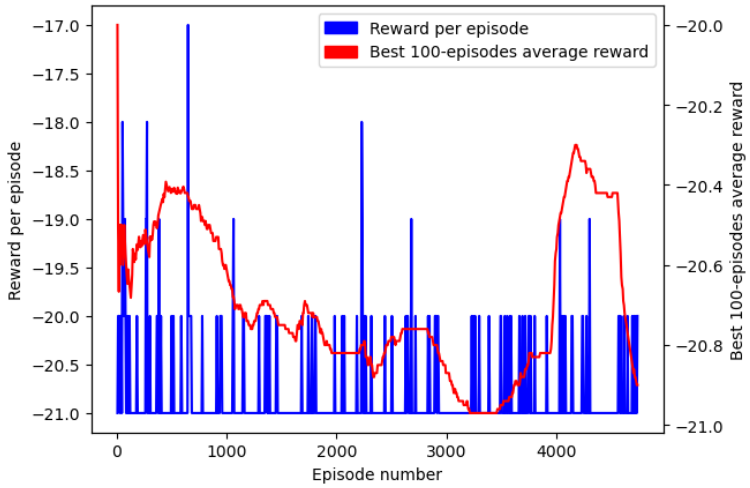
复现其他变体
复现 Double DQN：在主文件夹下运行 `python atari_ddqn.py --train`.
复现 Dueling DQN：在主文件夹下运行 `python atari_duelingdqn.py --train`.
复现 DQN with Prioritized Replay Buffer：在主文件夹下运行 `python atari_dqn_per.py --train`.

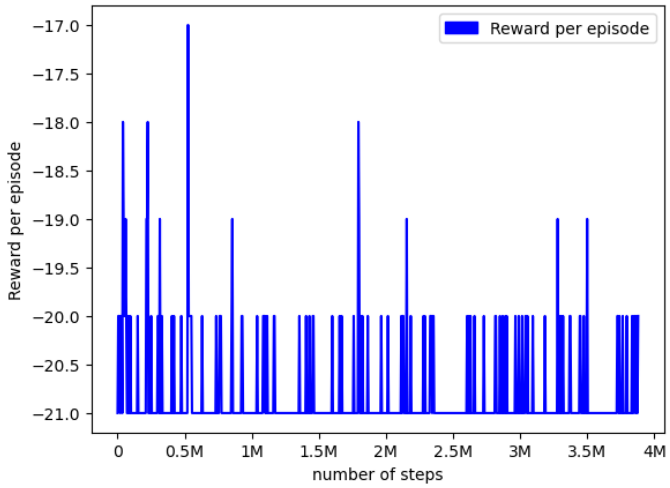
5 实验效果

在相同的超参数设置下，整体实验效果为 Double DQN 最终可以打赢对手，获得最高 20 分的奖励。观察到一些有意思的现象，智能体找到了一些对手的破绽，每次都会利用这些破绽进行一些精准打击。即和羽毛球一样，喜欢打边角球，比如对手在左上角，就往左上角打，但是由于墙体反弹，球实际落点是在左下角，对手来不及到达，就会输掉比分。

DQN 在经过更长时间的训练后仍未能赢得比赛，其每情节的奖励仍然在-19 分波动。

描述累计奖励和样本训练量之间的关系。
DQN:

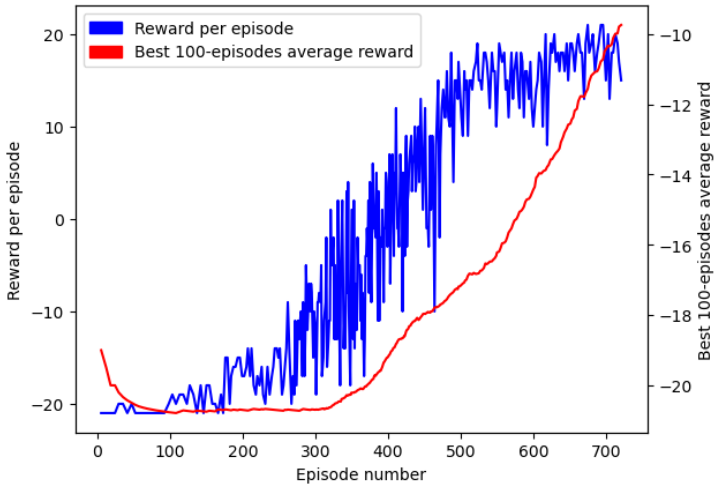


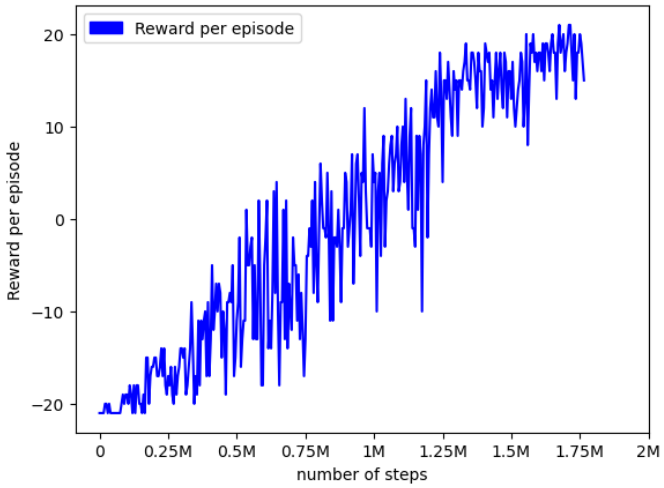


其他变体:

Double DQN:

```
TIME 06h 14m 03s  num timesteps 1750000, FPS 77
 Loss 0.001, avrage reward 18.2
TIME 06h 15m 14s  num timesteps 1755000, FPS 77
 Loss 0.001, avrage reward 18.1
TIME 06h 16m 25s  num timesteps 1760000, FPS 77
 Loss 0.002, avrage reward 18.7
TIME 06h 17m 32s  num timesteps 1765000, FPS 77
 Loss 0.000, avrage reward 18.7
TIME 06h 18m 42s  num timesteps 1770000, FPS 77
 Loss 0.001, avrage reward 18.2
Ran 722 episodes best 100-episodes average reward is 18.030000. Solved after 622 trials ✓
```





最后再描述一下两种模型的不同现象以及深层次的原因。可以看到，尽管超参数设置相同，DQN 由于其对 Q 值的过估计导致训练过程无法收敛，最终没能赢得比赛，比分一直在-19 分上下波动；而 Double DQN 针对 Q 值被高估的问题做出了相应改进，即选动作的 Q 函数与计算值的 Q 函数不是同一个，因此可以很快收敛到最优解，取得最高 20 比分的成绩。

6 小结

在这次实验中，我发现游戏环境相对上一次作业困难了很多，状态和动作空间都相对较大，即使使用深度神经网络在 GPU V100 上也需要训练 5、6 个小时才能收敛到不错的结果。

另外，算法的超参数调节也比较重要，一般可以先调节学习率，出现震荡现象可以把学习率调小，出现 loss 下降太慢现象可以将学习率调低。另外，通常情况下，Adam 优化器是个不错的选择，如果还是无法收敛，可以将一些中间结果的梯度信息输出到 TensorBoard 里观察，如果出现梯度方差极大且不规律的现象，可以进行梯度裁剪操作获取更稳定的结果。

最后，关于算法本身方面，DQN 本身是第一个将深度网络应用到强化学习并取得较好结果的模型，针对其不同的问题，后续算法分别进行了改进，Double DQN 针对 DQN 的过估计问题，使用一个网络选择动作，另一个网络输出对应 Q 值，缓解了过估计问题；Dueling DQN 从 Q 值分解的角度出发，将 Q 值分解为状态价值 V 和动作优势值 A；DQN with Prioritized Replay Buffer 算法从样本选择的角度出发，采用优先级采样的方式，倾向于抽取 TD 误差大的样本，加快训练速度。

但这些基础的改进和目前的 SOTA 比如 SAC 和 TD3 仍有不小的差距，目前的主流方式仍然是结合 Actor 和 Critic。面临复杂的工业场景，也需要更加鲁棒和稳定的强化学习模型，这些都需要我们进一步探索，总之，强化学习有着很大的发展空间，希望其能为人类带来更多的福祉。