DCS245
Reinforcement Learning and Game Theory
2021 Fall

Mid-term Assignment

19335025 陈禹翰
19335026 陈煜彦

0 背景

breakout

1 从马尔科夫链到 DQN

1.1 强化学习

强化学习考虑智能体**Agent**和环境**Environment**之间交互的任务，这些任务包含一系列的动作**Action**, 观察**Observation**还有反馈值**Reward**。智能体每一步根据当前的观察从动作集合中选择一个动作执行，目的是通过一系列动作获得尽可能多的反馈值。

1.2 MDP

马尔可夫决策过程中下一个状态仅取决于当前的状态和当前的动作。一个基本的 MDP 可以用 (S, A, P) 来表示，S表示状态，A表示动作，P表示状态转移概率，也就是根据当前的状态s_t和a_t转移到s_{t+1}的概率。状态的好坏等价于对未来回报的期望，引入回报**Return**来表示某个时刻t的状态将具备的回报，也就是 G_t。R 表示反馈，γ 是折扣因子。

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

价值函数用来表示一个状态的长期潜在价值，也就是回报的期望

$$v(s) = \mathbb{E}\left[G_t | S_t = s\right]$$

价值函数可以被分解为两部分，一个是立即反馈 R_{t+1}，还有一个是下一个状态的价值乘上折扣

$$\begin{aligned} v(s) &= \mathbb{E}\left[G_t | S_t = s\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma\left(R_{t+2} + \gamma R_{t+3} + \ldots\right) | S_t = s\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma G_{t+1} | S_t = s\right] \\ &= \mathbb{E}\left[R_{t+1} + \gamma v(S_{t+1}) | S_t = s\right] \end{aligned}$$

上面这个公式就是**Bellman**方程的基本形态，它描述了状态之间的迭代关系，说明当前状态的价值和下一步的价值以及当前的反馈**Reward**有关。

$$v(s) = R(s) + \gamma \sum_{s' \in S} P(s'|s) v(s')$$

1.3 Q-Learning

考虑到每个状态之后都有多种动作可以选择，每个动作之下的状态又多不一样，我们更关心在某个状态下的不同动作的价值。我们使用 Action-Value function 来表示在 s 状态下执行 π 策略之后获得的回报。

$$q_\pi(s, a) = \mathbb{E}_\pi\left[G_t | S_t = s, A_t = a\right]$$

它也可以被分解为

$$q_\pi(s, a) = \mathbb{E}_\pi\left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a\right]$$

现在，要寻找最优策略则可以等价于求解最优的 Action-Value function （当然，这只是其中的一种方法）。

$$\begin{aligned} q_*(s, a) &= \max_\pi q_\pi(s, a) \\ &= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a'} q_*(s', a') \end{aligned}$$

我们可以使用 Q-learning 的方式来求解。类似于 Value Iteration，Q-learning 更新 Q 值的方法如下：

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

具体算法如下：

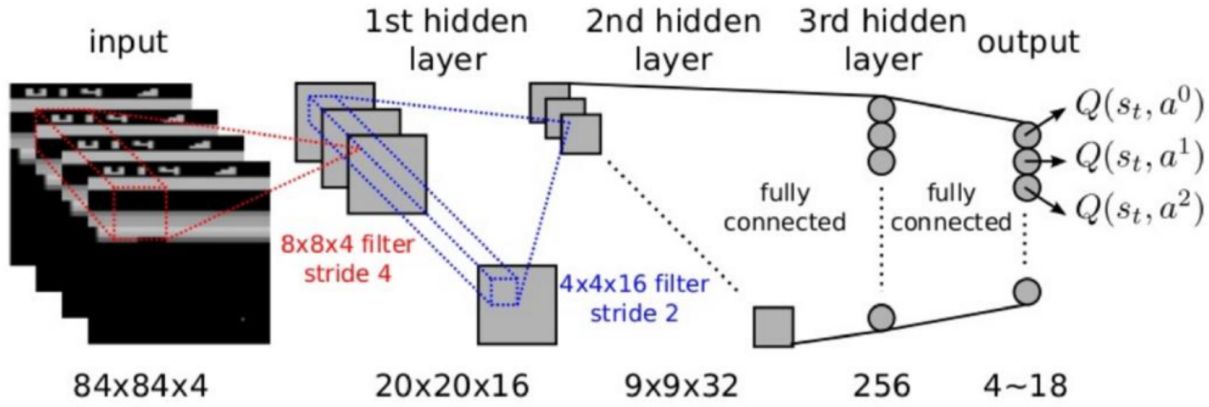
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$;
  until S is terminal

1.4 DQN

一般来说，我们会使用一个表格来存储 Q 值，就像之前做过的 Cliff Walking 作业一样。但是在此次作业要实现的 Breakout 中这个方法不太可行，因为数据实在是太大了，不可能通过表格来存储状态。如此大的数据也难以快速学习。因此我们使用 Action-Value Function Approximation 对状态的维度进行压缩。

在 Breakout 游戏中，状态是高维度的，而动作只有左移右移和不动。所以我们只需要对状态进行降维。输入一个状态输出一个状态与不同动作结合的向量。

由于输入的状态是四个连续的 84×84 图像，所以我们使用深度神经网络来表示这个降维的函数。具体由两个卷积层和两个全连接层组成，最后输出包含每一个动作 Q 值的向量。



接下来利用 Q-Learning 算法训练 Q 网络。在 Q-learning 中，我们利用 Reward 和 Q 计算出来的目标 Q 值来更新 Q 值，因此，Q 网络训练的损失函数就是：

$$L(w) = \mathbb{E}\left[\left(\underbrace{R + \gamma \max_{a'} Q(s', a', w)}_{Target} - Q(s, a, w)\right)^2\right]$$

NIPS 2013 提出的 DQN 算法如下：

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

由于玩 Breakout 采集的样本是一个时间序列，样本之间具有连续性，如果每次得到样本就更新 Q 值，受样本分布影响，效果会不好。因此，一个很直接的想法就是把样本先存起来，然后随机采样，也就是 Experience Replay。通过随机采用的数据进行梯度下降。

2 dqn-breakout 的结构解析

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

2.1 main.py

`mian.py` 是整个程序的入口，它首先定义了这些常量：

```
START_STEP = 0          # start steps when using pretrained model

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 = 0.05          # after decay steps, spsilon will reach this and keep
EPS_DECAY = 1_000_000   # 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 = 10_000 # evaluate frequency
```

- `START_STEP` 是模型开始训练的步数，方便使用已有的模型继续计算。没有使用预先训练好的模型开始计算时为 `0`；

- `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:4d} {step:8d} {avg_reward:.1f}\n")
    if RENDER:
        prefix = f"eval/eval_{step//EVALUATE_FREQ:04d}"
        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(f"models/model_{step//EVALUATE_FREQ:04d}")
    done = True
```

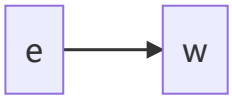
2.2 utils_drl.py

2.3 utils_env.py

2.4 utils_model.py

2.5 utils_memory.py

3 使用 Dueling DQN 提高性能



4 Experiments

5 总结

5.1

5.2 Open Source Repository

Our code and report are open source at lzzmm/breakout.

5.3 Authorship

Name	ID	Ideas(%)	Coding(%)	Writing(%)
陈禹翰	19335025	50%	40%	60%
陈煜彦	19335026	50%	60%	40%

References