强化学习
作业四

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

2022 年 12 月 28 日

1 作业内容

在 gridworld 环境中实现 Q-learning 算法.

2 实现过程

2.1 定义超参数

`args.num_step`	每次更新训练集采集的步数, default=100
`args.num_frames`	训练的总样本 (帧) 数, default=25000
`gamma`	QAgent 中更新 Q-table 时的超参数, default=0.9
`alpha`	QAgent 中更新 Q-table 时的超参数, default=0.1
`epsilon`	epsilon-greedy 中的超参数

2.2 实验探究一

首先实现 Dyna-Q 算法, 该算法伪代码如图1.

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \varepsilon$-greedy(S,Q)
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
- (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Loop repeat n times:
  - $S \leftarrow$ random previously observed state
  - $A \leftarrow$ random action previously taken in S
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

图 1: Dyna-Q 算法

这个算法简单明了, 前四步 (a,b,c,d) 和普通的 Q-learning 算法一样; 只有第 (e) 和 (f) 步稍微不同. 在 (e) 中, environment 的模型是基于确定环境下的假设 (对于非确定的环境或者是非常复杂的环境则根据特定的情况来做 assumption); (f) 步骤可以被概括为 Q-planning, 即使用已经学习到的模型来更新 Q 函数 n 次, 最后一行更新 Q-table 的方式和 (d) 步一样.

Qagent: 与 Q-learning 一样, Qagent 通过维护一个 Q-table 来评估不同 observation 下的不同 action, Qagent 的具体代码实现如下:

```
class QAgent:
    def __init__(self, envs, gamma=0.9, alpha=0.1):
        self.gamma = gamma
        self.alpha = alpha
        self.qtable = np.zeros((envs.image_size, envs.image_size, 2, envs.action_shape))

    def select_action(self, ob):
        q_values = self.qtable[tuple(ob)]
        action = np.argmax(q_values)
        return action

    def update(self, ob, action, reward, next_ob, done):
        sa = tuple(np.append(ob, action))
        target = reward + (1 - done) * self.gamma * self.qtable[tuple(next_ob)].max()
        self.qtable[sa] += self.alpha * (target - self.qtable[sa])
```

其中 `select_action()` 方法选取输入 observation 下 Q 值最大的 action; `update()` 方法用于更新 Q-table, 具体更新方法见算法1的 (d) 和 (f) 的最后一行.

DynaModel: 在 Q-learning 的代码上进行简单修改, 可以实现 Dyna-Q 的主要代码. 最主要的修改是加入了环境模型 DynaModel, 具体代码实现如下:

```
class DynaModel(Model):
    def __init__(self):
        Model.__init__(self)
        self.memory = {}

    def store_transition(self, s, a, r, s_):
        key = tuple(np.append(s, a))
        self.memory[key] = s_

    def sample_pair(self):
        choices = list(self.memory.keys())
        idx = np.random.choice(len(choices))
        return choices[idx][:-1], choices[idx][-1]

    def predict(self, s, a):
        search_key = tuple(np.append(s, a))
        return self.memory.get(search_key, None)
```

环境模型 DynaModel 用一个字典表示, 每次在真实环境中收集到新的数据, 就把它加入字典, `store_transition()` 方法用于存储采集到的 `((s, a), s_)` 对. 根据字典的性质, 若该数据本身存在于字典中, 便不会再一次进行添加. 在 Dyna-Q 的更新中, 执行完 Q-learning 后, 会立即执行 Q-planning. 在每轮 Q-planning 中, 使用 `sample_pair()` 对过往采集的 (s, a) 对进行采样, 使用 `predict()` 方法找到 `(s, a)` 对应的下一时刻 `s_`.

在 Dyna-Q 中, 同样的强化学习方法既可以用于从实际经验中学习也可以用于从模拟经验中进行规划, 每当 agent 采取一个 action 时, 学习的进程同时通过实际的选择的 action 和环境模型的模拟来更新, 这样就能够加快 agent 的学习速度. Dyna-Q 算法唯一的超参数是 Q-planning 进行的论述 `n`, 不同 `n` 下的实验效果见第 4 节.

2.3 实验探究二

2.3.1 实验 1

此方法与 Dyna-Q 的主要区别在于用 neural network 代替原本的字典查找, 使用一个 buffer 作为经验池, `store_transition()` 方法用于存储过往的经验 `[s, a, r, s_]`; `train_transition()` 方法则是利用经验池中的样本, 使用 neural network 根据 `(s, a)` 预测对应的 reward `r` 和 next state `s_`; `predict()` 方法使用训练过的 neural network , 根据 `(s, a)` 预测 reward `r` 和 next state `s_`, 用于后续每轮 Q-planning 中的模拟和 Q-table 的更新. `train_transition()` 的具体实现如下:

```
def train_transition(self, batch_size):
    s_list, a_list, r_list, s_next_list = [], [], [], []

    for _ in range(batch_size):
        idx = np.random.randint(0, len(self.buffer))
        s, a, r, s_ = self.buffer[idx]
        s_list.append(s)
        a_list.append([a])
        r_list.append(r)
        s_next_list.append(s_)

    s_tensor = torch.tensor(s_list).float()
    a_tensor = torch.tensor(a_list).float()
    s_next_tensor = torch.tensor(s_next_list).float()

    predict = self.network(s_tensor, a_tensor)
    loss = self.criterion(predict, s_next_tensor)

    self.optim.zero_grad()
    loss.backward()
    self.optim.step()
    return loss.item()
```

另外, 与 Dyna-Q 略有不同, 除了使用超参数 `n` 决定 Q-planning 的轮数, 还使用 epsilon-

greedy 对 `(s, a)` 进行采样, 并使用 neural network 预测 reward `r` 和 next state `s_`, 并更新 Q-table.

2.3.2 实验 2

在上述基于 neural network 的环境模型上进行改进, **改进 1** 将持有钥匙状态发生变化的样本作为更有价值的样本, 进行着重训练; **改进 2** 将每次更新后的 Q 值规约到 [-100, 100]. 两种改进的实验结果见 4.2.2.

3 复现方式

在主文件夹下运行 `python main.py` 即可. 若要运行 Dyna-Q 算法, 在 `main.py` 的 `execute()` 中令 `dynamics_model = DynaModel()`, 并注释掉下一行.

若要运行基于 neural network 的 Model-based 算法, 则要令 `dynamics_model = NetworkModel(8, 8, policy = agent, args = args)`, 并注释掉上一行.

若要运行改进 Model 的学习流程, 则要令 `dynamics_model = NetworkModel(8, 8, policy = agent, args = args)`, 并在主文件夹下运行 `python main.py --modify1 True`.

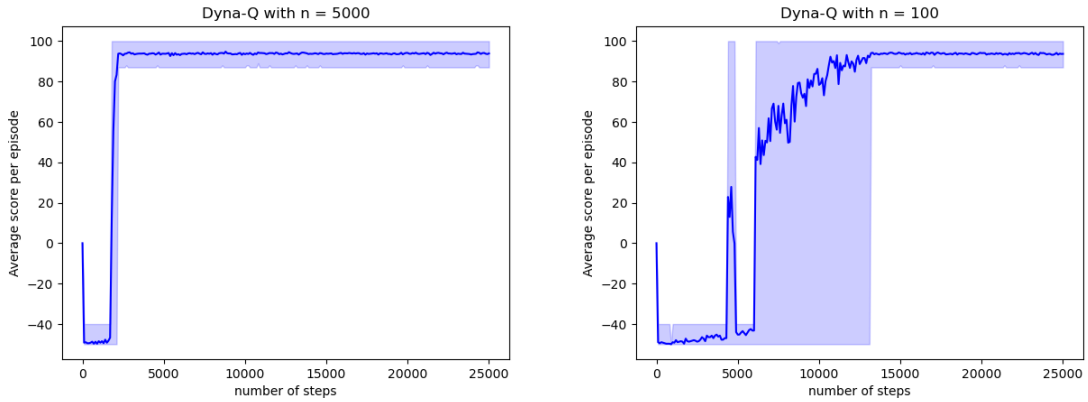
4 实验效果

4.1 实验探究一

选取不同的超参数 `n`, 实验结果见下表:

n	100	300	1000	2000	3000	5000	6000
time(s)	11	8	3	2	3	**2**	4
number of samples (k)	13	9.4	3.4	2.8	3.4	**2.2**	3.4

可以选出最优的超参数 `n*`=5000, 需要 2s 和 2.2k 样本即可收敛. `n`=5000 时和 `n`=100 时的平均奖励随样本数量的变化曲线见图4.1.



从上述结果中我们可以很容易地看出, 随着 Q-planning 步数的增多, Dyna-Q 算法的收敛速度也随之变快. 当然, 并不是在所有的环境中, 都是 Q-planning 步数越大则算法收敛越快, 这取决于环境是否是确定性的, 以及环境模型的精度.

基于模型的强化学习算法 Dyna-Q 在 gridworld 环境就可以获得很好的效果, 因为该环境中的状态和动作都是离散的, 模型可以很容易通过经验数据得到.

4.2 实验探究二

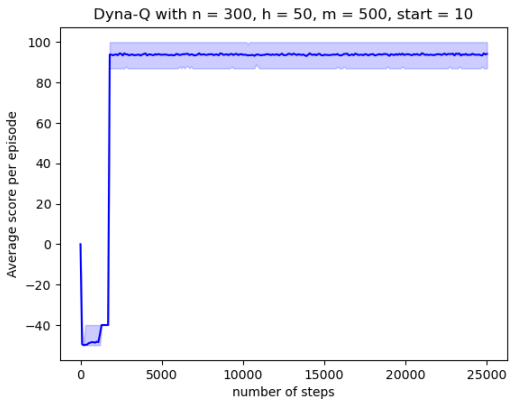
4.2.1 实验 1

本实验采用 grid search 方法搜索最优的参数组合, 可以调整的参数包括:

- `n`: 进行 Q-planning 的轮数, 经过一些粗糙的尝试, 将进行 grid search 的范围固定在 `[100, 300, 500]`;
- `h`: 每轮 Q-planning 中采样的轮数, 进行 grid search 的范围 `[10, 30, 50]`;
- `m`: 根据经验池中的样本训练 neural network 的轮数, 进行 grid search 的范围 `[100, 300, 500]`;
- `start_planing`: 开始进行 Q-planning 的步数, 进行 grid search 的范围 `[5, 10, 30]`.

```
params = {
    'n': [100, 300, 500],
    'h': [10, 30, 50],
    'm': [100, 300, 500],
    'start_planning': [5, 10, 30]
}
for vals in product(*params.values()):
    execute(**dict(zip(params, vals)))
```

经过在不同超参数组合上的 grid search, 得到模型收敛最快, 需要样本数最少得参数组合为 `n=300, h=50, m=500, start_planning=10`, 在样本数量约为 2.1k 时收敛, 此时平均奖励随样本数量的变化曲线见图4.2.1.



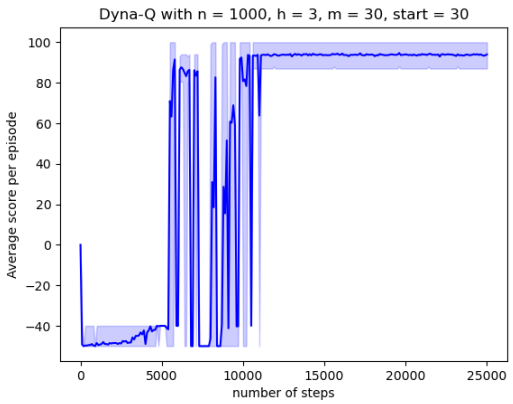
下表展示了一些实验结果差异较大的超参数组合, 其中有一些参数组合是 grid search 之前的一些尝试中的结果. 根据这些尝试和 grid search 的结果, 我得出以下结论:

1. `n` 不再像前一个实验, 越大越好, 最佳范围是几百; `h` 不宜太大, 因为使用 epsilon-greedy 进行采样, 样本容易重复, 采样轮数过多没有意义; `m` 取值范围和 `n` 一样, 多训练 neural network 有利于环境模型拟合样本, 但数值不宜过大, 早期过拟合会适得其反; `start_planing` 应取较小的值, 尽早进行 Q-planning;

2. 比较本次实验的平均奖励曲线与 Dyna-Q 的奖励曲线, 可以发现在模型收敛的过程中 Dyna-Q 的奖励曲线上升更为稳定, 而基于 neural network 的奖励曲线在收敛之前的波动更为明显, 例如图4.2.1, 由于 gridworld 环境状态为离散且较为简单, 因此在这里 Dyna-Q 展示出更强的鲁棒性.

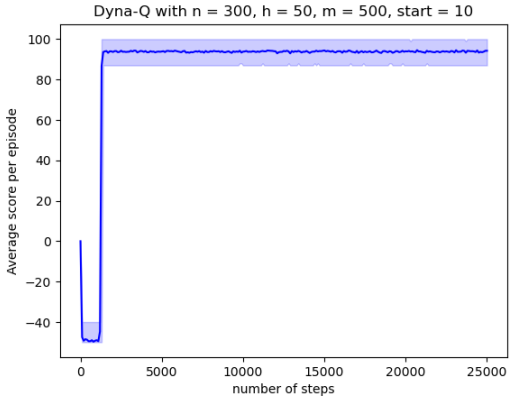
3. 虽然 `n` 的最佳取值范围和实验探究 1 中 `n` 的最佳取值相差很多, 但其真正的采样数量为 `n*h`, 其数量级和实验探究 1 中 `n` 类似, 甚至更大, 说明基于 neural network 的 model-based 算法需要更多的样本, 其样本效率不如基于 table 的 model-based 算法.

n	100	300	300	300	500	500	500	1000	3000
h	10	20	30	**50**	20	20	10	10	10
m	50	50	100	**500**	100	50	30	30	30
start_planning	15	20	10	**10**	10	30	10	100	10
number of samples (k)	7.6	5.7	4.3	**2.1**	4.3	6.2	8.7	11.5	8.9

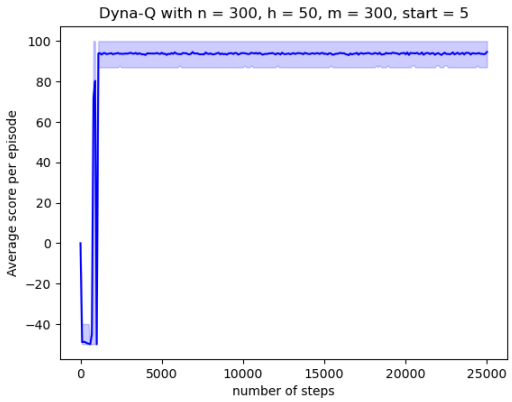


4.2.2 实验 2

改进 1: 经过改进 1, grid search 得到的最优参数组合依旧是 `n=300, start_planning=10, h=50, m=500`, 在样本数量约为 1.4k 时收敛, 此时平均奖励随样本数量的变化曲线见图4.2.2. 相较上一实验, 此参数组合依旧是性能最优的参数组合, 且模型性能在改进后也获得了提升.



改进 2: 在每次更新 Q-table 时, 将更新后的 Q 值规约到 [-100, 100], grid-search 的结果表明之前取得最优的参数组合不再是最优的, 最优的参数组合为 `n=300, start_planning=5, h=50, m=300`, 在 0.9k 处收敛. 改进 2 中的最优参数组合和之前的最优参数组合相差很小, 其奖励变化曲线见图4.2.2.



另外, 在改进 2 中, 所有参数组合的整体性能和其他实验相比更好, 且能取得较好性能 (在 1.5k 前收敛) 的参数组合更多.

4.3 实验探究 3

开放问题:

1. 从不同模型学习方式看, 在 gridworld 这样相对简单的环境中, table 比 neural network 能更好帮助算法提升样本效率. 尽管两者都能很快收敛, 但基于 table 的环境模型在收敛的过程中更加稳定.

   这是因为 gridworld 是离散环境, 且环境是确定性的, 环境状态和可选动作数量都不是特别大, table 的表示能比 neural network 更快更准确地学得环境模型. 同时, 以 neural network 作为环境表征时, 其训练次数 `m` 必须足够大, 否则无法提升算法的采样效率. 这些都说明环境模型的准确度对 model-based 算法有很大影响. 另外, Dyna-Q 算法生成的额外训练样本必须足够多, 才能显著提升算法性能. 这可以归结为对环境模型的充分利用.

2. 与 DQN 中的 replay buffer 设置对比, 两者的联系在于 Dyna-Q 实验也相当于是对 replay buffer 作了改进: 引入了更多更 on-policy 的样本. 由此可见正确修改 replay buffer 的数据分布有助于提升强化学习算法的样本效率.

5 小结

在本次作业中, 尝试了基于 table 和基于 neural network 的 model-based 方法, 两者均取得了较好的性能. 但在 gridworld 这样简单的环境中, table-based model 能取得更好的样本效率, 且收敛较稳定. 这是因为 gridworld 是离散环境, 且环境是确定性的, table 的表示能比 neural network 更快更准确地学得环境模型. 最后, 实验结果表明影响实验效果的因素不仅包括环境是否是确定性的, 还在于环境模型的精度.