强化学习：作业四

丁云翔 191250026

2021 年 12 月 29 日

1 作业内容

在 gridworld 环境中实现 Dyna-Q 算法，并通过调节参数找到算法可提升的性能极限；用神经网络来预测环境 Model，实现简单的 Model-based 算法，完成三个探究问题。

2 实现过程

实验探究 1：

将 main.py 中的 dynamics_model 初始化修改为 DynaModel 类型变量。policy 学习部分使用了作业 2 中的内容，不再赘述，都位于 algo.py 的 MyQAgent 类中。在 main.py 中需要加入 Q-Learning 的位置加入了 agent.learn 函数的调用。实现了 algo.py 中的 DynaModel 类，同样使用 table 来进行记录和更新，具体实现如下：

```python
class DynaModel(Model):
    def __init__(self, width, height, policy):
        Model.__init__(self, width, height, policy)
        self.buffer = []
        self.qTable = defaultdict(lambda: [0.0, 0.0, 0.0, 0.0])

    def store_transition(self, s, a, r, s_):
        self.buffer.append([s, a, r, s_])

    def sample_state(self):
        idx = np.random.randint(0, len(self.buffer))
        return self.buffer[idx][0], idx

    def sample_action(self, s):
        return self.policy.select_action(s)

    def predict(self, s, a):
        max = 0.0
        s_list= [self.sample_state()[0]]
        for item in self.buffer:
            if str(item[0]) == str(s) and item[1] == a:
                if item[2] > max:
                    s_list.clear()
                    s_list.append(item[3])
                    max = item[2]
                elif item[2] == max:
                    s_list.append(item[3])
        return random.choice(s_list)

    def train_transition(self):
        s, a, r, s_ = random.choice(self.buffer)
        oldQ = self.qTable[str(s)][a]
        newQ = r + self.policy.discountFactor * max(self.qTable[str(s_)])
        self.qTable[str(s)][a] += self.policy.learningRate * (newQ - oldQ)
```

实验探究 2：

修改算法参数进行测试，并基于文档给出的改进方法对算法进行改进。

3 复现方式

实验探究 1：

将 main.py 中的 dynamics_model 初始化修改为 DynaModel 类型变量，然后在主文件夹下运行 `python main.py`.

实验探究 2：

将 main.py 中的 dynamics_model 初始化修改为 NetworkModel 类型变量，然后在主文件夹下运行 `python main.py`.

4 实验效果

实验探究 1：

n=0 时，大约在 97 轮达到收敛，收敛时间约为 33s，消耗的样本量约为 9800。

n=1 时，大约在 83 轮达到收敛，收敛时间约为 29s，消耗的样本量约为 8400。

n=2 时，大约在 83 轮达到收敛，收敛时间约为 28s，消耗的样本量约为 8400。

n=3 时，大约在 87 轮达到收敛，收敛时间约为 29s，消耗的样本量约为 8800。

n=4 时，大约在 90 轮达到收敛，收敛时间约为 29s，消耗的样本量约为 9100。

n=5 时，大约在 91 轮达到收敛，收敛时间约为 29s，消耗的样本量约为 9200。

n=6 时，大约在 87 轮达到收敛，收敛时间约为 28s，消耗的样本量约为 8800。

n=7 时，大约在 93 轮达到收敛，收敛时间约为 30s，消耗的样本量约为 9300。

由此可以发现，当 $n > 1$ 后，算法收敛所消耗的样本量不再有明显的下降。

实验探究 2：

n=0、m=0、start_planning=0、h=0 时，大约在 102 轮达到收敛，收敛时间约为 36s，消耗的样本量约为 10300。

n=1、m=0、start_planning=0、h=0 时，大约在 92 轮达到收敛，收敛时间约为 33s，消耗的样本量约为 9300。

n=1、m=1、start_planning=10、h=1 时，大约在 77 轮达到收敛，收敛时间约为 27s，消耗的样本量约为 7800，但在后续依然存在很多波动。

n=1、m=2、start_planning=10、h=2 时，大约在 132 轮达到收敛，收敛时间约为 46s，消耗的样本量约为 13300，但在后续依然存在很多较大幅度的波动。

n=3、m=3、start_planning=10、h=3 时，大约在 561 轮达到收敛，收敛

时间约为 3m32s，消耗的样本量约为 56200，但在十余轮后又开始大幅度波动，最后跑到 999 轮还没有再次达到收敛状态。
n=3、m=5、start_planning=20、h=10 时，跑了 999 轮还没收敛，收敛时间大于 6m14s，消耗的样本量大于 100000。
由此可以发现，综合收敛速度与稳定性而言，n=1、m=0、start_planning=0、h=0 时效果最好。

尝试改进 1 后再测试:
n=0、m=0、start_planning=0、h=0 时和 n=1、m=0、start_planning=0、h=0 时的性能可直接参考原来的表现，因为 m=0 时 train_transition 不会被调用，所以改进不会对该参数下对应的性能产生影响。
n=1、m=1、start_planning=10、h=1 时，大约在 102 轮达到收敛，收敛时间约为 34s，消耗的样本量约为 10300，但在后续依然存在很多波动。
n=1、m=2、start_planning=10、h=2 时，大约在 110 轮达到收敛，收敛时间约为 42s，消耗的样本量约为 11100，但在后续依然存在很多较大幅度的波动。
n=3、m=3、start_planning=10、h=3 时，大约在 122 轮达到收敛，收敛时间约为 43s，消耗的样本量约为 12300，但在之后又开始大幅度波动，不过能够再次达到收敛状态。
n=3、m=5、start_planning=20、h=10 时，跑了 999 轮还没收敛，收敛时间大于 6m16s，消耗的样本量大于 100000。
由此可以发现，综合收敛速度与稳定性而言，仍然是 n=1、m=0、start_planning=0、h=0 时效果最好，对应的性能变化不大，因为此参数下，train_transition 不会被调用，所以改进不会对该参数下对应的性能产生影响。其他参数下的性能总体上略有提高，收敛速度更快一些。

尝试改进 2 后再测试:
n=0、m=0、start_planning=0、h=0 时，大约在 86 轮达到收敛，收敛时间约为 33s，消耗的样本量约为 8700。
n=1、m=0、start_planning=0、h=0 时，大约在 84 轮达到收敛，收敛时间约为 31s，消耗的样本量约为 8500。

n=1、m=1、start_planning=10、h=1 时，大约在 97 轮达到收敛，收敛时间约为 37s，消耗的样本量约为 9800，但在后续依然存在很多波动。
n=1、m=2、start_planning=10、h=2 时，大约在 98 轮达到收敛，收敛时间约为 36s，消耗的样本量约为 9900，但在后续依然存在很多较大幅度的波动。
n=3、m=3、start_planning=10、h=3 时，大约在 195 轮达到收敛，收敛时间约为 1m10s，消耗的样本量约为 19600，但在十余轮后又开始大幅度波动，最后跑到 999 轮还没收敛，收敛时间大于 6m18s，消耗的样本量大于 100000。
n=3、m=5、start_planning=20、h=10 时，跑了 999 轮还没收敛，收敛时间大于 6m42s，消耗的样本量大于 100000。
由此可以发现,综合收敛速度与稳定性而言,仍然是 n=1、m=0、start_planning=0、h=0 时效果最好，改进后对应的性能有所提升，收敛更快。其他参数下的性能也有所提高，尤其是收敛速度有明显提高。

关于问题的答案：
1. 不同模型学习方式会导致模型的复杂度不同，不同参数会影响算法对 model 的依赖程度。较复杂的模型训练的收敛速度较慢，但收敛后对环境的建模效果会更好；较简单的模型收敛速度较快，但收敛后对环境的建模效果较差。start_planning 决定了从多少轮开始基于学得的 model 进行模拟，若开始太早，可能学得的模型误差较大，对 policy 的更新造成不利；若开始太晚，则 model 的价值不大。n 和 h 决定了每一轮对 model 的利用程度，n 和 h 越大，policy 的学习对 model 就越依赖，若 model 误差小，则 policy 性能会提升；若 model 误差大，则 policy 性能可能会下降。m 决定了每轮对 model 的训练程度，m 越大，每轮对 model 的训练轮数就越大，若存储的转移数据较多，则 model 的效果会更好；若存储的转移不够多，则训练轮数大会导致 model 的过拟合。因为最后还是要对 policy 进行学习，与 policy 学习相关的参数也会对性能造成影响，如 epsilon、alpha、gamma。
故影响 model-based 算法性能的因素有 policy 学习相关的参数、model 的学习方式（复杂程度）、对 model 进行转移训练的频率、使用 model 进行模拟采样的轨迹条数、每条轨迹的执行长度、开始使用 model-based 模拟的轮

数。

2. Dyna-Q 中的 buffer 是为了对模型进行模拟，并用来在模拟与环境的交互中对自身的 Q 函数和策略进行预测和更新；DQN 中的 bufffer 是为了减弱时间序列之间的相关性，让采样数据分布更稳定，从而可以让算法多次使用样本。二者的联系在于都可以看作是经验，都存储了与环境交互的历史数据，都可以用来对动作进行预测，都可以提高数据的利用效率，都可以使算法用更少的数据得到更好的性能。

5 小结

在这次实验中，我发现基于 model 的强化学习在这个场景下效果并不算好，而且模型越复杂，效果就越差。我认为原因可能是这个场景的环境并不复杂，若用过于复杂的模型来模拟环境，一来收敛速度慢，而来可能会导致过拟合。另外一个原因是开始轮数太早，对环境的探索还不完全，若以此对环境进行建模可能会导致比较大的误差，为了验证这一原因我也将 start_planning 调到过 50 进行了测试，发现收敛速度略有提高，但后续依然存在较大波动。若继续上调 start_planning 意义也不大，因为最好的结果在 80 多轮就能收敛了，就算收敛速度继续提高也优势不大，反而是稳定性问题依然无法解决。