作业

Homework4

庄镇华 502022370071

A RL Homework Assignment

2022 年 12 月 21 日

实验环境

本次作业使用和作业 2 一样的环境为网格世界 (gridworld)，玩家可以通过选择动作来移动人物，走到出口。唯一的区别在于输出的状态包括了额外的一维特征，表示 agent 是否拿到了钥匙。agent 需要先拿到钥匙（坐标在（0,7)），然后走到出口才算通关。

问题 1

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。

实验要求：

1. 写完代码后，请从 n=0 开始（即纯 model-free 方法），尝试调试不同的参数 n，记录算法的收敛时间，和所消耗的样本量。得出一个经验性的 n^* 的粗略估计，表示若 n 的取值 $n > n^*$ 算法收敛所消耗的样本量不再有明显的下降。

2. 请在实验报告中展示你所尝试的参数和对应的实验结果。

问题 2

用神经网络来预测环境 Model，实现简单的 Model-based 算法，完成以下三个探究问题

实验 1：算法调试

1. 该实验的 Model 相关接口及其实现已经写好，调节算法的参数，寻找你能找到的达到最好效果的参数组合 a. n (采样的轨迹条数)，b. start_planning (开始使用 model based 提高样本利用率)，c. h （一条轨迹执行的长度）d. m （转移训练的频率）e. ... 其他你发现的有影响的参数

2. 请在实验报告中展示你所尝试的有显著差异的参数组合和实验结果

实验 2：改进算法

改进 1：尝试改进 Model 的学习流程，强化对稀疏/奖励变化相关的数据的学习。

改进 2：对策略的学习过程做额外的约束：

分别尝试两个改进，重新调节该探究问题中实验 1 的参数组合，最优的参数和对应的性能是否发生变化？若有变化，发生了什么变化？

（Optional）：可以尝试其他任意的改进，并展示你的改进带来的性能提升。

问题 3

根据上面的实验回答以下两个问题（开放问题）

1. 根据上面实验，试讨论不同模型学习方式（table 和 neural network），不同参数对实验结果的影响和背后的原因，从而分析影响 model-based 的算法的性能的因素由哪些？有以下两条参考建议

a. 可打印学习过程的以下中间指标辅助进行分析：Q 函数学习情况如何？策略表征如何？模型在各个状态的预测准确度如何？

b. 可回顾老师上课提到的 Model-based 相关的三个问题进行思考，即：how to learn the model efficiently? how to update the policy efficiently? how to combine model learning and policy learning?

2. 回顾 HW3 的 DQN 中的 replay buffer 设置和前面的 Dyna-Q 实验，你觉得这两者有什么联系?

解答:

1 作业内容

a. 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。

b. 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。

c. 根据实验结果分析影响 Model-based 算法性能的因素，说明 DQN 中的 replay buffer 设置和前面的 Dyna-Q 实验的联系。

2 算法原理

2.1 Dyna-Q

算法 1 Dyna-Q

对于 $\forall s \in S, a \in A(s)$，初始化 $Q(s,a)$ 以及 $Model(s,a)$
repeat
  $S \leftarrow$ 当前状态
  $A \leftarrow \varepsilon$-贪婪算法 (S,Q)
  执行动作 A; 观察到奖励 R 和状态 S'
  $Q(s,a) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
  $Model(S,A) \leftarrow R, S'$(假设是确定性环境)
  for i ∈ range(n) **do**
    $S \leftarrow$ 随机选择之前观察过的状态
    $A \leftarrow$ 随机选择之前在该状态下执行的动作
    $R, S' \leftarrow Model(S,A)$
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
  end for
until 算法收敛

Dyna-Q 算法融合了 Model-based 和 Model-free 的强化学习方法，缓解了 Model-based 方法因为不能精确拟合环境而导致效果很差的问题。它既在模型中学习，也在交互中学习。

即 Dyna-Q 方法在每次迭代中先和环境交互，并更新价值函数、策略函数，接着进行 n 次模型的模拟预测，同样更新价值函数、策略函数。这样同时利用了和环境交互的真实样本以及模型的预测样本，可以利用更少的交互样本达到和 Model-free 方法类似的效果。

3 实现过程

3.1 实验探究 1

首先针对 main.py 修改，将 dynamics_model 初始化为 DynaModel。policy 初始化为作业 2 中实现的 QLearningAgent，并在相应位置加入 agent.update 函数的调用。

```
    agent = QLearningAgent(alpha, gamma)  # @zhuangzh
```

```
            agent.update(obs, action, reward, obs_next) # @zhuangzh
```

然后针对 algo.py 修改，QLearningAgents 实现如下：

```
class QLearningAgent(QAgent):
  def __init__(self, alpha, gamma):
    super().__init__()
    self.lr = alpha
    self.gamma = gamma
    self.qtable = defaultdict(lambda : [0., 0., 0., 0.])

  def select_action(self, ob):
    all_q = np.array(self.qtable[str(ob)])
    idxes = np.argwhere(all_q == np.max(all_q))
    return random.choice(idxes)[0]

  def update(self, ob, a, r, next_ob):
    old_q = self.qtable[str(ob)][a]
    new_q = r + self.gamma * max(self.qtable[str(next_ob)])
    self.qtable[str(ob)][a] += self.lr * (new_q - old_q)
```

使用 table 来进行记录和更新状态转移，DynaModel 实现如下：

```
class DynaModel(Model):
    def __init__(self, width, height, policy):
        Model.__init__(self, width, height, policy)
        self.transitions = []

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

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

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

    def predict(self, s, a):
        ns_list = [tran[-1] for tran in self.transitions if str(tran[0]) == str(s)
     and tran[1] == a]
        ns_list = ns_list or [s] # 考虑未出现的状态
        return random.choice(ns_list)

    def train_transition(self):
        pass
```

3.2 实验探究 2

需要修改的算法参数如下：

```
epsilon = 0.2
alpha = 1
gamma = 0.9999
n = 1000 # 采样的轨迹条数
m = 1000 # 转移训练的频率
start_planning = 10 # 开始使用model based 提高样本利用率
h = 1 # 一条轨迹执行的长度
```

改进 1 的代码如下，即改进 Model 的学习流程，强化对稀疏/奖励变化相关的数据的学习。

```
def store_transition(self, s, a, r, s_):
    s = self.norm_s(s)
    s_ = self.norm_s(s_)
    self.buffer.append([s, a, r, s_])
    if s[-1] - s_[-1] != 0:
        self.sensitive_index.append(len(self.buffer) - 1)
    # ''' 改进1 新增部分
    if s[-1] - s_[-1] != 0:
        self.sensitive_index.append(len(self.buffer) - 1)
    # '''
    # ''' 改进1 新增部分
    if len(self.sensitive_index) > 0:
        for _ in range(batch_size):
            idx = np.random.randint(0, len(self.sensitive_index))
            idx = self.sensitive_index[idx]
            s, a, r, s_ = self.buffer[idx]
            s_list.append(s)
            a_list.append([a])
            r_list.append(r)
            s_next_list.append(s_)
    # '''
```

改进 2 的代码如下，即对策略的学习过程做额外的约束。

```
            agent.update(obs, action, reward, obs_next) # @zhuangzh
            ''' 改进2 新增部分
            agent.qtable[str(obs)][action] = np.clip(agent.qtable[str(obs)][action
], -100, 100)
            '''
```

4 复现方式

4.1 实验探究 1

首先将 main.py 中的 dynamics_model 初始化为 DynaModel，然后调节参数 n 并在主文件夹下运行 python main.py.

4.2 实验探究 2

首先将 main.py 中的 dynamics_model 初始化为 NetworkModel，然后调节相应参数并在主文件夹下运行 python main.py.

5 实验效果

首先说明实验设置，实验收敛的**判别标准**为：当平均奖励大于 92 分，并且最大奖励和最小奖励差值小于 15 分时，认为此次实验收敛。

并且为了**排除随机性**的影响，实验所有数据都是以相同配置运行三次取平均得到。

5.1 实验探究 1

实验中固定的超参数有：

epsilon = 0.2 # 贪婪探索的系数

alpha = 1 # 学习率

gamma = 0.9999 # 衰减系数

m = 0 # 转移训练的频率

start_planning = 0 # 开始使用 model based 提高样本利用率

h = 1 # 一条轨迹执行的长度

n	0	1	2	3	4	5	6	7	8	9	10
收敛时间/s	20.3	28.3	31.3	45.7	57.3	62	60.3	66.7	59.7	70.0	78.2
消耗样本量/个	4233	4167	4167	4000	4000	4067	3767	3767	3400	3533	3600

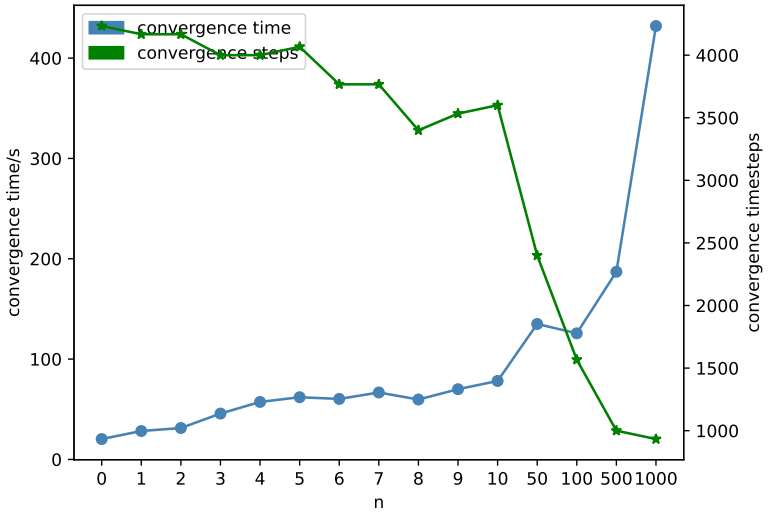
可以发现，总体趋势为随着 n 的增加，额外训练的次数越多，收敛时间越来越长，但消耗的样本量越来越少。但当 n 取较小值时，当 $n \geq 8$ 后，消耗的样本量不再减小，因此可以认为经验性的 $n^{\star} = 8$。

n	50	100	500	1000
收敛时间/s	135	126	187	432
消耗样本量/个	2400	1567	1000	933

可以发现，当 n 取较大值时，消耗的样本量仍会明显减小，但收敛时间会大大增加，这可能是因为环境为确定性转移，并且实验中 h 取值为 1，因此模型模拟的样本都是精确的，不会有误差累积，因此效果会持续提升，但其收敛时间也会随之增长，由于时间限制，没有探究 $n > 1000$ 后的情况。

n 的取值对收敛时间和消耗样本量的影响如图1所示，可以直观的看到，当 n 取较小值时，当 $n = 8$ 时，消耗的样本量达到最小值，但随着 n 的急速增加，消耗样本量仍然会减少，但此时收敛时间太长，因此不适合实际使用。

图 1: n 的取值对收敛时间和消耗样本量的影响



5.2 实验探究 2

5.2.1 算法调试

n	m	start_planning	h	收敛时间	消耗样本量	备注
0	0	0	0	23	3567	
8	1	10	1	44.2	6933	
8	2	10	2	62	9783	
8	3	10	3	134	20633	不稳定
8	3	20	1	43	7900	
8	3	30	1	41.7	7667	
8	50	10	1	45.7	7367	
8	100	10	1	32.3	3533	
8	100	20	1	36.7	5467	
8	100	30	1	31.67	4733	
1000	1000	10	1	288.3	8033	
1000	1000	20	1	91	3867	
1000	1000	20	2			2 万步后无法收敛
1000	1000	20	5			2 万步后无法收敛
1000	1000	30	2	110	4433	不稳定

总体而言，消耗样本量最少的超参数为 $n = 8, m = 100, start\_planning = 10, h = 1$，消耗的样本量为 3533。

针对最优参数的选择，由于有多个超参数需要调节，因此我们采取逐个击破的方法，即固定其他变量，只分析其中的一个变量。

首先分析 h，即一条轨迹执行的长度，可以发现，h 值一旦超过 1，就会生成很多模型没有见过的样本，即现有分布之外的样本，针对这些样本，即使模型在现有样本上训练的很

好（比如 $m = 1000$ 时，此时模型过拟合现有样本），模型给出的下一步状态随机，因此很容易导致无法收敛或不稳定的结果，在基于模型的方法前提下，h 的最优值为 1。

其次分析 n，即采样的轨迹条数，实验探究 1 已经得出当 n 较小时，最好的 n 为 8，因此我们先从 $n = 8$ 开始，实验发现 n 越大，则相对应最优的 m 也应该越大，这是因为当模型采样的轨迹条数增多，为了学习这些样本，相应的训练频率也要提高。并且，当 $n = 8$, $m = 100$ 或者 $n = 1000$, $m = 1000$ 时，模型都有不错的表现。

然后分析 m，即训练频率，可以发现，对于特定的 n，只有当 m 取适当的值，模型才会拟合好现有的样本，m 过小导致欠拟合，m 过大导致过拟合，其效果都不能达到最优。并且，当 $n = 8$, $m = 100$ 或者 $n = 1000$, $m = 1000$ 时，模型都有不错的表现。

最后分析 *start_planning*，即开始使用 model based 提高样本利用率的迭代时机，可以发现，这类似于半监督问题，模型采样的轨迹样本可以理解为无监督样本，即其标签是伪标签（即有可能标签是错误的），当 *start_planning* 值取得过小，模型还未来得及学习到好的表示就要面对很多伪标签干扰，而当 *start_planning* 值取得过大，最终迭代次数和收敛样本量也会增加。并且，当模型采样的轨迹条数 n 变大时，对应的最优 *start_planning* 值也会变大，可以解释为当模型采样的轨迹条数占总体样本比例变大时，需要利用 model free 方法积累到足够好而多的样本，才能应对使用 model based 方法带来的伪标签干扰。

5.2.2 改进算法

改进 1

改进 Model 的学习流程，强化对稀疏/奖励变化相关的数据的学习

n	m	start_planning	h	收敛时间	消耗样本量	备注
0	0	0	0	25	3600↑	
8	1	10	1	49	6000↓	
8	2	10	2	46	5700↓	
8	3	10	3	70	9900↓	
8	3	20	1	73	10500↑	
8	3	30	1	54	8200↑	
8	50	10	1	69	8200↑	
8	100	10	1	57	4300↑	
8	100	20	1	56	4200↓	
8	100	30	1	61	4700↓	
1000	1000	10	1	218	5800↓	
1000	1000	20	1	101	3400↓	
1000	1000	20	2	425	9400↓	
1000	1000	20	5			2 万步后无法收敛
1000	1000	30	2	668	12400↓	

最优的参数和对应的性能发生变化。总体而言，消耗样本量最少的超参数变为 $n = 1000, m = 1000, start\_planning = 20, h = 1$，消耗的样本量为 3400，比之前有了提升。

在其他参数下，模型收敛速度和性能几乎都比改进前要优越，由于对策略学习过程做了额外的约束，消除了学习过程中噪声的影响，因此模型更加稳定。

改进 2

对策略的学习过程做额外的约束

n	m	start_planning	h	收敛时间	消耗样本量	备注
0	0	0	0	32	3800↑	
8	1	10	1	64.2	7200↑	
8	2	10	2	39	4200↓	
8	3	10	3	61	5700↓	
8	3	20	1	42	4300↓	
8	3	30	1	37	4200↓	
8	50	10	1	75	5800↓	
8	100	10	1	63	3900↑	
8	100	20	1	86	4800↓	
8	100	30	1	77	4600↓	
1000	1000	10	1	321	8100↑	
1000	1000	20	1	99	3300↓	
1000	1000	20	2			2 万步后无法收敛
1000	1000	20	5			2 万步后无法收敛
1000	1000	30	2	110	3700↓	

最优的参数和对应的性能发生变化。总体而言，消耗样本量最少的超参数变为 $n = 1000, m = 1000, start\_planning = 20, h = 1$，消耗的样本量为 3300，比之前有了提升。

并且在其他参数下，大多数情况下，模型收敛速度和性能都比改进前要优越，值得注意的是，由于强化了对稀疏奖励变化相关数据的学习，模型的稳定性和收敛性比之前要好。

5.3 开放问题

问题 1

根据上面实验，试讨论不同模型学习方式（table 和 neural network），不同参数对实验结果的影响和背后的原因，从而分析影响 model-based 的算法的性能的因素由哪些？有以下两条参考建议

a. 可打印学习过程的以下中间指标辅助进行分析：Q 函数学习情况如何？策略表征如何？模型在各个状态的预测准确度如何？

b. 可回顾老师上课提到的 Model-based 相关的三个问题进行思考，即：how to learn the model efficiently? how to update the policy efficiently? how to combine model learning and policy learning?

答：不同模型的学习方式导致模型的复杂度不同，table 和 neural network 学习方式各有优缺点，table 模型学习的收敛速度较快，学习的环境转移为确定性方式，但对复杂环境建模能力差，鲁棒性弱，因而适合简单环境的学习；neural network 模型学习的收敛速度慢，但学习到的环境转移鲁棒性好，对复杂环境建模能力好，因而适合复杂环境的学习。

不同参数会影响算法对 model-based 方法中模型的依赖程度。start_planning 决定从多

少轮开始利用学到的模型模拟样本，其过小，可能导致学到的模型噪声过大，误差过大；其过大，则迭代次数过多，没有意义。

m 为每轮对模型的训练次数，当经验样本较多时，m 越大，则模型学习到的表示会更好；当经验样本不足时，训练轮数过大会导致模型的过拟合。

h 为一条轨迹执行的长度，h 值过大，就会生成很多模型没有见过的样本，即现有分布之外的样本，容易导致无法收敛或不稳定的结果。n 和 h 决定了每一轮对模型的依赖程度，n 和 h 越大，策略就对模型越依赖，如果模型误差小，则策略性能会提升；若模型误差大，则策略的性能会下降。

最后，epsilon、alpha、gamma 等参数也会直接影响策略本身的学习。

总结而言，影响 model-based 算法性能的因素有很多，具体而言，epsilon、alpha、gamma 等参数以及模型表示的方式 (table 或者 neural network) 会直接影响策略本身的学习，m、n、h 等参数通过影响模型训练的次数和模型训练的样本影响模型本身的学习，start_planning、n、h 等参数影响策略学习和模型学习的 trade_off，只有当调节好所有参数，model-based 算法才能取得较高的性能。

问题 2

回顾 HW3 的 DQN 中的 replay buffer 设置和前面的 Dyna-Q 实验，你觉得这两者有什么联系?

答：由于强化学习只能获取很少量样本中的信息，因此需要反复榨取样本的利用价值，HW3 的 DQN 中的 replay buffer 是直接把以往的经验样本存储起来，以供之后重复利用；而 Dyna-Q 则是利用自己的模型模拟环境转移，自己生成样本，并将之用于策略学习。

具体而言，DQN 中的 replay bufffer 可以减弱时间序列之间的相关性，稳定采样数据的分布，而 Dyna-Q 中的 buffer 可以对模型进行模拟，让算法利用更多的样本对策略进行学习更新。他们都存储了与环境交互的历史数据，都可以提高数据的利用效率，即用更少的数据获得更好的性能。

6 小结

在这次实验中，我发现 model-based 的强化学习算法可以利用更少的样本学到和 model-free 强化学习算法类似的性能，在无法试错且交互成本高昂的工业环境有很广泛的应用价值。但是，model-based 方法也有一定的局限性，比如时间复杂度过高，模型的学习可能存在误差，导致策略不稳定等。

并且，model-based 方法会受到很多超参数的影响，其存在三个基本问题，即 how to learn the model efficiently? how to update the policy efficiently? how to combine model learning and policy learning? 因而只有在模型学习误差较小、策略学习较优、并且模型学习和策略学习的 trade-off 调节好的前提下，model-based 方法才能取得较好的结果。