作业4:FreeWay游戏

张祎扬 （181840326 181840326@smail.nju.edu.cn)

(南京大学 匡亚明学院， 南京 210046)

摘要：使用强化学习来自主玩FreeWay游戏，通过Q值函数进行决策。尝试修改特征提取方法和强化学习的参数以求获得更好的学习性能。

关键词：强化学习、Q-Policy、特征提取、reward

1.task1

1.1 阅读代码，阐述强化学习的方法和过程。

强化学习让计算机从一开始完全随机地进行操作，通过不断尝试，从错误中学习，最后找到规律，学会了达到目的的方法。让计算机在不断的尝试中更新自己的行为，从而一步步学习如何操控自己的行为得到高分。它主要包含四个元素，Agent，state（状态），action（行动），reward（奖励），并且强化学习的目标就是获得最多的累计奖励。所以强化学习具有分数导向性。

下面具体解释框架代码。

QPolicy

首先进行初始化,初始化action的数量，数据集，随机数，并把classifier初始化为空。

```
public QPolicy(int N_ACTIONS){
  m_numActions = N_ACTIONS;
  m_dataset = RLDataExtractor.datasetHeader();
  m_rnd = new Random();
  m_c = mull;
}
```

方法`getQArray()`的作用是首先创建一个double类型的数组，大小是m_numActions，然后用一个for循环给这个double数组按索引赋值，赋-的值是预测模型返回的Q值。

方法`fitQ()`的参数是data，如果m_c为空，则相应地对m_c进行处理，用一个叫做`REPTree()`的决策树来对m_c进行初始化。如果m_c不为空，则直接进入最后一步，调用weka包的`buildClassifier()`方法，把data作为训练集训练分类器。

方法`makeInstance()`的作用就是用m_dataset建立新的Instance对象。

在QPolicy.java中，最重要的两个方法是`getActionNoExplore()`和`getAction()`，这两个方法的主要区别是是否应用epsilon greedy策略。首先分析二者代码的相同部分。

```
double[] Q = getQArray(feature);
//find best action according to Q value
int bestaction = 0;
for(int action=1;action<m_numActions;action++){
  if(Q[bestaction] < Q[action]){
    bestaction = action;
  }
}
```

以上代码的作用是，首先定义一个double数组Q，用来存放当前局面的Q值，接着用一个for循环遍历Q数组，找到Q值最大的索引，返回该索引作为最佳的动作（即对应的Q值最大的动作）。

```
//among the same best actions,choose a random one
int sameactions=0;
for(int action=bestaction+1;action<m_numActions;action++){
  if(Q[bestaction]==Q[action]){
    sameactions++;
    if(m_rnd.nextDouble()< 1.0/(double)sameactions)
      bestaction = action;
  }
}
```

这一段代码的作用是，假如有多个action索引的Q值相同，都是最大值，那么就随机选择一个。由于之前遍历的for循环的条件是<，所以这就决定了找到的bestaction一定是同样Q值的action中的第一个，所以在这段代码中，遍历只需要从action+1开始.如果发现了Q值一样的action，就以$\frac{1}{n}$的概率决定要不要替换掉原来的bestaction.其中，n的值是sameaction,也就是Q值相同的action的个数。这也预示着，随着发现的Q值相同的action越来越多，替换的概率也越来越小。

`getActionNoExplore()`的代码到这里就结束了，它返回bestaction.但是`getAction()`的代码还多出来一部分如下，这一部分就是epsilon greedy的算法。

```
//epsilon greedy
if(m_rnd.nextDouble() < m_epsilon){
  bestaction = m_rnd.nextInt(m_numActions);
}
```

m_epsilon的值决定了执行探索的概率。执行探索就是随机选择一个动作返回，`getAction()`每次都以m_epsilon的概率执行一次探索。假如不执行探索，就执行利用，利用就是返回bestaction.m_epsilon的设置和`getActionNoExplore()`最大的区别就在于，它不是一味执行利用，而是通过设置m_epsilon的值，并且可以调整这个值，来平衡探索和利用的关系。当数据集较少，不确定性更大时，可以执行更多的探索；反之可以执行更多的利用。

Agent

首先在Agent方法中是一系列初始化。

`simulate()`方法中，首先设置了一个最大迭代深度`SIMULATION_DEPTH`,在`simulate()`函数中，它尝试在当前的局面中往下走`SIMULATION_DEPTH`步，记录每走一步之前的`score_before`和走完该步之后的`score_after`，并且计算`delta_score`。

```
double delta_score = factor*(score_after-score_before);
factor = factor*m_gamma;
```

`delta_score`的计算有factor作为权重，并且每更新一次模拟，factor的值就乘以m_gamma.在每一次模拟中，将计算得到的`delta_score`值按顺序存入数组sequence中。

接着从最后一次的状态中用`getMaxQ()`方法获取预测的Q值，并且存入变量accQ中。然后再计算累计的Q值，并且随着深度递减一步步往回更新accQ值，并将其存入data中，最后返回data。

在`learnPolicy()`方法中，首先调用`simulate()`方法获取数据集(即所有累计的Q值)，然后更新数据集。

```
//update dataset
m_dataset.randomize(m_rnd);
for(int i =0;i<dataset.numInstance();i++){
  m_dataset.add(dataset.instance(i));//add to the last
}
while(m_dataset.numInstances()>n_maxPoolSize){
  m_dataset.delete(index:0);
}
```

这一段代码的作用是限制用来训练的数据集的大小，利用n_maxPoolSize()来限制最大的数据集的大小，如果大小超过了它，就删除数据集的第一条数据。这里在更新数据时采用了覆盖的方式，来保证数据集即不超过限制的大小，又不断处于更新状态。

最后针对更新过的数据集，调用`firQ()`方法进行训练。

最后解释`act()`类，它返回一个动作来让Agent作出反应。它调用了上面介绍过的`learnPolicy()`方法,针对更新的数据集训练模型。然后它调用`getActionNoExplore()`，也就是没有探索地寻找Q值最大的动作作为bestaction，并且返回它。最终agent就返回了一个动作。

以上是对代码的分布解释，下面整体介绍Q-learning算法。

Q-learning 就是创造一个Q表，来指导智能体的行动，Q表对应Action的数值越大，智能体就越大概率地采取这个Action。根据Q表的值采取行动只是最终的步骤，Q表的获得是经过学习之后的结果，学习并不断更新Q表的过程就是Q-Learning.

在Q-Learning学习并更新Q表的过程中，可以使用epsilon贪婪方法，来设置一个epsilon值用来控制执行探索的概率，这样的方法可以通过适当地调整epsilon的值来平衡探索和利用，从而达到更好的学习性能。

Q-Learning算法的关键在于如何更新Q表的值。当智能体处于S_0当状态时，如果刚好选择了A_2,那么下一个状态就是S_1。行动之后，就需要更新$Q(S_0, A_2)$的数值。

$$Q(S_0, A_2) = Q(S_0, A_2) + \alpha[R(S_1) + \gamma * max_a Q(S_1, a) - Q(S_0, A_2)]$$

$R(S_1)$是智能体在S_1状态的奖励（reward），γ为衰减值，$max_aQ(S_1,a)$是S_1状态下Q表数值最大的一个，α是学习速率，$R(S_1)+\gamma*max_aQ(S_1,a)$是$Q(S_0,A_2)$的目标数值。

所以以上公式可以表示为，这很类似线性回归中的梯度下降法。

$$Q(S_0,A_2)_{新}=Q(S_0,A_2)_{旧}+\alpha*[Q(S_0,A_2)_{目标}-Q(S_0,A_2)_{旧}]$$

完整的公式如下：

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

从以上的公式可以看出，S_t和S_{t+1}是一个递归的关系，当智能体走到n步的时候，会受到0到n-1每一步状态的影响，设置一个γ值在0到1之间，可以让越靠近n的状态对第n步影响越大，越早对状态影响力会越来越小。

以上参考了书本和网站[1-4].

1.2 策略模型用什么表示？该表示有何缺点？有何改进方法?

策略模型用Q-Learning表示，该表示存在过高估计的问题。因为Q-Learning在更新Q函数时使用的是下一时刻最优值对应的action，这样会导致过高估计采样过的action。

可以使用Sarsa算法，它与Q-Learning的不同在于，处于状态s'时，知道了要采取哪个a',并且真的采取了这个行动，目标Q值的计算也是根据策略得到的动作a'计算而来。而在Q-Learning算法中，仅计算了在状态s'时采取哪个a'可以得到更大的Q值，并没有真的采取这个动作a',目标Q值的计算是根据Q值最大的动作a'计算而来。因为Sarsa算法采取了实际的行动再回传Q值，所以相比之下并没有采取实际行动的Q-Learning就有一点与实际情况脱离，它只通过计算最佳Q值来选择动作，也可能会出现过度估计的问题。

以上回答参考了网站[5-6].

1.3 Agent.java 代码中 SIMULATION_DEPTH, m_gamma, m_maxPoolSize 三个变量分别有何作用?

1. `SIMULATION_DEPTH`变量的作用是限制模拟的最大深度。将它设定为一个合适的值，既可以防止时间和计算的消耗过大，又可以在一定程度上保证模拟的精度。在框架代码中，变量的初始值是20，就是意味着在当前的局面的基础上往下模拟20步，并且根据这20步的结果更新Q值表。
2. `m_gamma`变量就是以上代码介绍中提到的衰减值，框架代码中将它设置为0.99.因为每次更新Q值都要乘以m_gamma,所以越远离当前局面的动作的Q值改变量的权重越低，这也体现了Q-Learning算法中越近的状态对Q值更新影响越大。
3. `m_maxPoolSize`变量的作用在上面介绍框架代码的时候已经分析过。在框架代码中它的值被设置为1000。它限制了数据集的大小，当数据集的大小达到`m_maxPoolSize`的值的限制的时候，每更新一条数据集，就删除数据集中索引为0的第一条，这样就以一个覆盖的方式保证了数据集始终是更新的，但是它的大小又不超出限制。

1.4 QPolicy.java 代码中，getAction 和 getActionNoExplore 两个函数有何不同？分别用在何处?

关于这个问题，我在上面分析框架代码的时候也有涉及。这两个函数的代码大体相同，唯一的区别是`getAction()`比`getActionNoExplore()`多了一段如下代码：

```
//epsilon greedy
if(m_rnd.nextDouble() < m_epsilon){
  bestaction = m_rnd.nextInt(m_numActions);
}
```

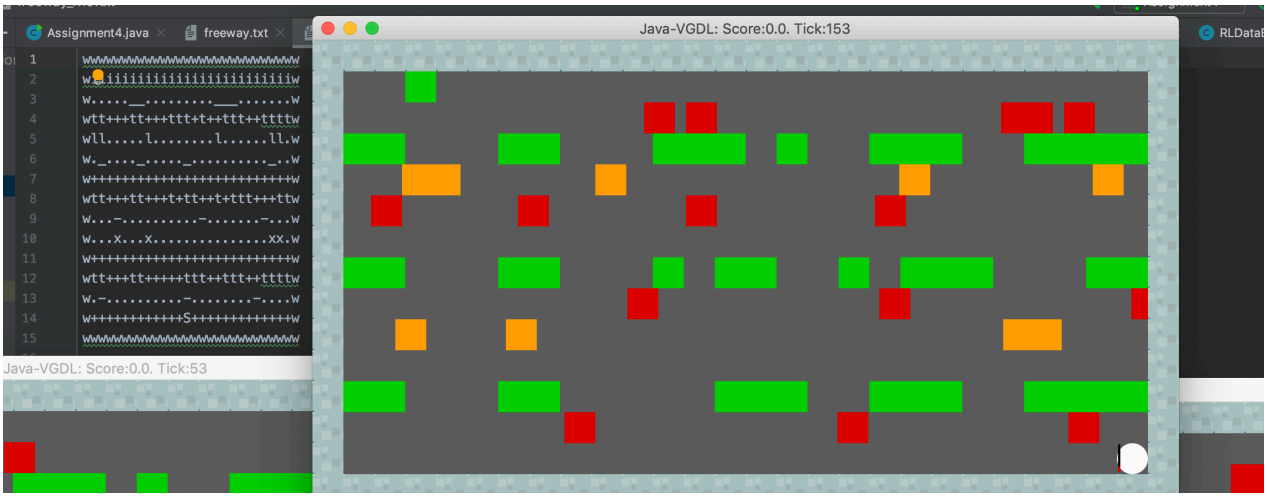
这段代码就是采用了epsilon greedy策略，通过设置epsilon值来设置探索和利用的概率。以epsilon的概率来进行探索，也就是随机选择一个action作为bestaction返回，以1-epsilon的概率进行利用，返回Q值最大的action作为bestaction。`getActionNoExplore()`因为没有这一步，所以它不进行探索，是完全进行利用的。

相比之下，`getActionNoExplore()`因为缺乏一些随机性，更容易陷入局部最优解中，它适用于不确定性较小的情景，不需要什么探索，只要利用就可以得到比较好的解。而`getAction()`通过设置m_epsilon值，应用epsilon greedy策略，在一定程度上对探索和利用进行了平衡，我们还可以通过自己设置m_epsilon的值来平衡利用和探索。当不确定性较大时，可以更多的进行探索，增加随机性。当不确定性较小，概率分布较集中时，不需要太多探索就可以获得比较优的解，所以可以减少探索的概率，多进行利用。

2. task2

尝试修改特征提取方法，得到更好的学习性能，并报告修改的尝试和得到的结果.

首先，我在特征提取的函数中尝试输出`o.itype`,发现它大致有6种：0，7，8，10，11，13.然后我仔细研究了第0关的地图和规则描述，如下图：



可以看到它确实和规则中描述的一样。然后我修改了特征提取函数，在给feature赋值的循环中输出x和y的值和它的itype值，然后根据地图比对，来推断各type分别是什么。

经过比对我发现13是绿色的挡板，11是左移的红色慢车，10是左移橙色快车，8是右移的红色慢车，7是右移的橙色快车，0是地面。

这里并没有get到avatar的位置，入口的位置。所以我上官网[7]看了对`StateObservation`类的方法的简介,发现除了特征提取函数中用到的方法以外，还有一些方法也是很有用的。`getGameScore()`方法可以获取当前局面的得分。`getAvatarPosition()`方法返回avatar的位置。`getPortalsPosition()`返回的是终点（也就是目标入口）的位置。`getResourcesPositions()`返回的是财富的位置（我不知道这是什么意思但是启发式函数heuristic中使用了这个）。所以我将这些添加到代码中。添加后的代码如下。

```
LinkedList<Observation> allobj = new LinkedList<>();
if( obs.getImmovablePositions()!=null )
    for(ArrayList<Observation> l : obs.getImmovablePositions()) allobj.addAll(l);
if( obs.getMovablePositions()!=null )
    for(ArrayList<Observation> l : obs.getMovablePositions()) allobj.addAll(l);
if( obs.getNPCPositions()!=null )
    for(ArrayList<Observation> l : obs.getNPCPositions()) allobj.addAll(l);
if( obs.getPortalsPositions()!=null )
    for(ArrayList<Observation> l : obs.getPortalsPositions()) allobj.addAll(l);
if( obs.getResourcesPositions()!=null )
    for(ArrayList<Observation> l : obs.getResourcesPositions()) allobj.addAll(l);
```

之后我重新运行并观察输出，发现type多了一类4。4只出现了一次，对照地图可知显然是目标的位置。

阅读特征提取函数，可以知道feature数组的前$28 * 31 = 868$个分别是28*31的地图上的元素的type，接下来的四个分别是游戏的时间，avatar的速度，avatar的健康值和avartar的type。最后两个分别是动作和reward（也就是Q值）。我觉得这些对于学习是不够的。

在对它修改的过程中，我添加了对游戏得分的记录，通过调用`getGameScore()`方法。

```
feature[874] = obs.getGameScore();
```

要想增加学习的效果，还可以记录avatar到目标的距离远近(根据游戏的特殊性，主要是垂直距离，即y的差值）。所以我还添加了一个元素用来记录avatar到目标的距离。根据之前的推测，已知目标的type值为4，所以当type值为4时，记录目标的位置。相应的代码如下：

```
//get portalsposition
int portal_x = 0;
int portal_y = 0;
for(int y=0; y<31; y++)
  for(int x=0; x<28; x++){
    feature[y * 28 + x] = map[x][y];
    if(map[x][y] == 4){
      portal_x = x;
      portal_y = y;
    }
  }
//get avatar position
Vector2d avatar_position = obs.getAvatarPosition();
int avatar_x = (int)(avatar_position.x/28);
int avatar_y = (int)(avatar_position.y/28);
//get the distance in y
double distance_avater_portal = Math.abs(avatar_y-portal_y);
//add the feature
feature[875] = distance_avater_portal;
```

但我觉得还应该提升avatar躲避车子的能力，因为为了达到目标，它需要不断往上走，并且躲避来往的车辆，所以我决定记录avatar前后共三行的车辆（即与avatar的垂直距离小于等于1）与avatar的水平距离。这里面其实又涉及到了一个分类，因为车子的走向是固定的，如果车子往右走，我认为只需要记录avatar和它左边的第一辆车辆的距离。如果车子往左走，只需要记录avatar和它右边的第一辆车的距离。一部分代码如下：

```
//往左边寻找向右的第一辆车
for(int x = avatar_x;x>=0;x--){
  if(map[x][avatar_y-1] == 8 || map[x][avatar_y-1] == 7){
    cardis_up=avatar_x-x;
    break;
  }
}
//往右边寻找向左的第一辆车
for(int x = avatar_x;x<28;x++){
  if(map[x][avatar_y-1] == 11 || map[x][avatar_y-1] == 10){
    cardis_up = x-avatar_x;
    break;
  }
}

//record
feature[876] = cardis_up;
feature[877] = cardis_on;
feature[878] = cardis_down;
```

修改完之后我尝试让它学习，发现它倾向于一直卡在最右边，不能往上走，所以我突然意识到还必须记录一下avatar离最近的corssing的距离。当avatar的上一个（即x相同，y=y-1）位置处是挡板（type=13）时，记录avatar离最近的corssing的距离（从avatar向两边寻找）。具体代码如下：

```
double distance_crossing = 0;
if(map[avatar_x][avatar_y-1] == 13){
    double disleft = 0;
    double disright = 0;
    //find towards left
    for(int x=avatar_x;x>=0;x--){
        if(map[x][avatar_y-1]!=13){
            disleft = x-avatar_x;
            break;
        }
    }
    //find towards right
    for(int x = avatar_x;x<28;x++){
        if(map[x][avatar_y-1]!=13){
            disright = x-avatar_x;
            break;
        }
    }
    distance_crossing = (Math.abs(disleft)<=Math.abs(disright))? disleft : disright;
}

```

为了保证一个比较长的学习过程，我把healthPoint设置成了10（原来是5），我还把时间限制设置成了5000。

在修改完特征提取函数之后，我发现还是没办法使avatar通过学习自己跑到对面去，不过它往对面走和躲避车辆的行为趋势显著增加了。原来avatar基本一直在底部的角落里徘徊，现在它的行为的不确定性增加了，而且它经常会出现越过好几层挡板的行为。期望在修改了参数以后它的性能可以提高吧.....

3. task3

尝试修改强化学习参数，得到更好的学习性能，并报告修改的尝试和得到的结果

可以用来调整的参数有`m_epsilon`,`SIMULATION_DEPTH`，`m_gamma`。我花了很多时间去调整这些参数，却没有观察到avatar的行为有什么特别显著的变化。所以我决定继续修改reward的计算方法。

有关的代码是WinScoreHeuristic.java中的`evaluateStates()`函数。通过观察可以发现原本的函数只是通过最终的状态是win还是lose赋值。这显然是不够的。

在游戏中，avatar的目标是不断向入口靠近，并且不被车撞到。这里面涉及两个参数，一个是avatar和入口的距离，距离越大分数越低。我给它赋权重为6.另一个参数是avatar的生命值，越高越好，所以直接相加即可，我给它赋权重为5.相应的代码如下：

```
//get the y-distance  between avatar and the y=0
Vector2d avatar_pos = stateObs.getAvatarPosition();
int avatar_x = (int)(avatar_pos.x)/28;
int avatar_y = (int)(avatar_pos.y)/28;
//System.out.println(avatar_y);
//rawScore+=60*(14-avatar_y);
double distance = 0;
int health = stateObs.getAvatarHealthPoints();
//System.out.println(health);

LinkedList<Observation> allobj = new LinkedList<>();
if( stateObs.getPortalsPositions()!=null )
    for(ArrayList<Observation> l : stateObs.getPortalsPositions())
allobj.addAll(l);
for(Observation o : allobj){
    Vector2d p = o.position;
    int x = (int)(p.x/28);
    int y= (int)(p.y/28);
    distance = Math.abs(avatar_y-y)+Math.abs(avatar_x-x);
}
rawScore-=6*distance;
rawScore+=5*health;
```

然后再次进行游戏，我发现avatar有了更为明显的往目标处前进的趋势。但还是不能到达对面，总是在不停徘徊或者撞到车辆。于是我又对一些特殊位置做了调整。比如我更倾向于avatar穿过挡板，所以当avatar处在挡板一行时，它可以有一定的分数奖励。还有就是为了避免车辆撞到，我更希望avatar在没有车辆的那一行，所以也给予一定的奖励。修改后的代码如下：

```
if(avatar_y==3||avatar_y==7||avatar_y==11)rawScore+=3;
if(avatar_y==1||avatar_y==6||avatar_y==10)rawScore+=2;
```

修改完以后，我又进行了尝试，发现avatar还是不能非常顺利地到达对面。于是我又开始了漫长的调参之路……我不断调整上面提到的那些参数。最终因为时间有限还是没有达到非常理想的结果。

我把m_epsilon值调整到了0.5，提高探索的比例，在一定程度上增加了动作的随机性。然后我把m_gamma的值调整到了0.9，适当减小权重。鉴于游戏的复杂度不是特别高，我又把SIMULATION_DEPTH的值调整到了10.在这种情况下，avatar可能会出现到达对面的情况，但是它的行为还是有很大的波动性，也有可能它一直在下面波动，或者上升到一定高度之后撞到车辆于是回到下方。我尝试继续调整参数使它达到一个比较稳定的能通关的状态，但是并没有取得很好的效果。

这样的原因可能是多方面的，首先就是强化学习需要大量的数据和训练时间。然后就是可能我的特征提取函数还不足够提取到一些比较关键的特征。再可能就是调整参数的过程进行得还不够全面，没有进行更多的尝试，以及reward的设计上可以有更多的思考。

4. 结束语

本次作业是一次对强化学习的简单探索，在作业的过程中，我了解了强化学习的基本原理，重点了解了Q值算法。同时，我还经历了漫长而痛苦的调参过程，在这一过程中，我充分体会了特征提取和奖赏函数对于强化学习性能的重要性，也更深刻地体会了各参数在其中的作用与影响。

致谢：

References:

[1]http://reinforcementlearning.ai-depot.com/

[2]https://www.jianshu.com/p/f8b71a5e6b4d

[3]《Artificial Intelligence: A modern approach》chapter 21 Reinforcement Learning

[4]https://www.jianshu.com/p/1db893f2c8e4

[5]https://blog.csdn.net/zchang81/article/details/77746313

[6]https://www.zhihu.com/question/280077512

[7]http://www.gvgai.net/forwardModel.php