Pysc2\_maddpg项目工作报告

目录

1.主要工作汇报1

1.1算法改进2

1.1.1 价值网络（Q函数）评估的改进3

1.1.2 策略网络（μ函数）和价值网络（Q函数）参数更新顺序改进4

1.1.3 奖励函数（reward函数）的改进4

1.2代码改进6

1.2.1 动作空间改进6

1.2.2 初始动作改进7

1.3 地图条件改进7

1.3.1 地图设置改进7

1.3.2 不同阵型改进8

1.4 不同算法、阵型、初态对比测试8

2.遗留问题10

2.1 控制帧的问题10

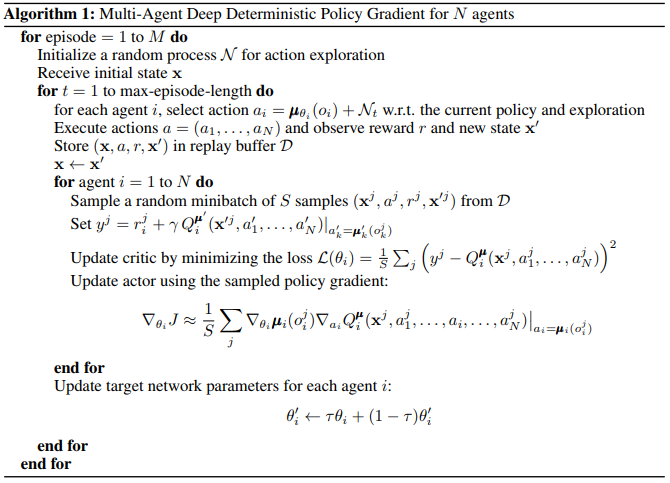
2.2 奖励函数（reward函数）的问题11

3.针对遗留问题的设想方案11

**1.主要工作汇报：**

**1.1算法改进：**

游戏中操作士兵的算法主体还是使用maddpg算法，但是针对maddpg算法进行了如下几处改进和调优。首先贴出maddpg的算法的伪代码：



**1.1.1** **价值网络（Q函数）的评估改进：**

在基于强化学习的算法中一直存在一个问题就是对于Q函数的值更新不准确的问题，并且如果Q函数更新不准确，那么对于算法中一直想逼近的Q函数最大值也会出现错误，比如在某个场景x下，应该采取动作a才是Q函数的最佳值，但是因为评估不准确，可能此时采取动作a’得出的Q是最大的，那么因为μ函数是一直朝向Q函数提升方向所更新的，所以会导致Q函数使得μ函数给出的策略朝a’动作来进行，但是实际上我们所希望的最佳策略是朝a动作来进行，这样会导致整个网络最后的策略不是最佳的。

在伪代码中的问题即为Q的评估不准确带来的若干问题，所以我们可以考虑将在以target为目标更新的时候，将Q中的参数a更改为μi(x’j)来解决此问题，这样会避免我们在错误的Q函数上一直进行下去带来训练不准确，训练时间变长的问题，并且同时有利于我们对于Q函数的正确评估。

在代码中这个改变的体现为在maddpg.py的update函数中将target\_act\_next\_n改为如下：

target\_act\_next\_n = [*agents*[i].p\_debug['p\_values'](obs\_next\_n[i]) *for* i *in* range(self.n)]

**1.1.2** **策略网络（μ函数）和价值网络（Q函数）参数更新顺序改进：**

在maddpg算法中，我们可以发现关于Q和μ的参数更新顺序为先更新各个Q函数的参数，然后计算最新的μ函数梯度，然后更新μ函数的参数，这样我们可以最新的Q来计算μ函数，使得更新更加快速。但是在原始代码中，对于μ函数和Q函数的参数却并不完全遵从算法，这样会导致我们μ函数的估计并非最新，使得μ函数更新速度变慢。因此我们修改算法代码，将参数更新顺序调优，并且配合1.1.1中Q函数评估的改动，让更近准确的Q函数来引导μ函数更新，使得正确率可以进一步上升。

在代码中这个改变的体现为在maddpg.py的update函数中将参数调优顺序改为如下：

q\_loss = self.q\_train(\*(obs\_n + act\_n + [target\_q]))  
self.q\_update()  
p\_loss = self.p\_train(\*(obs\_n + act\_n))  
self.p\_update()

**1.1.3** **奖励函数（reward函数）改进：**

在扩充动作空间，并且修改初始动作后，原来reward只简单粗暴将前后两帧的游戏内得分相减的思想显然已经不适用，因为如果继续使用这个reward方式，我们很容易就可以理解我们所操作的agent甚至不会主动靠近敌人，从而会在游戏开始时造成随机行动导致分散的情况，这令我们想到势必要修改reward的方式。

首先我们希望agent是有索敌的想法的，所以对于我们agent和敌人之间距离的变化我们应该建立一个reward来反映这个问题，同时我们应该注意以下两点：（1）当我们的agent与target的距离小于agent的攻击范围时，此时再降低敌我距离不应该带来reward，因为这时候攻击才是正确的选择。（2）当我们的agent血量减少的时候，我们也应该有一个reward来反映这件事情。（3）最重要的是我们还应该有一个纠缠项，要使得总reward在以下条件满足这样的大小关系：当agent低血量选择拉近距离时，其再度受伤获得的总reward相对于此时拉开距离获得的总reward为负。这样才可以满足我们agent避免盲目拉近距离送死的场面出现。

针对问题（1）初步确立的reward\_distance函数为后一帧距离敌方各个士兵的距离减去5后与0取最大值之和减前一帧距离敌方各个士兵的距离减去5后与0取最大值之和的差再除以-50（其中5为士兵攻击距离，负号保证拉近后reward\_distance取正，50为平衡reward\_distance之间数量级的参数）

针对问题（2）设想的是针对agent血量再设置一个reward\_hp函数，并且这个reward\_hp函数满足，当agent血量越低时，agent掉血得到的reward\_hp越小。

针对问题（3）设想总reward应该为a\*reward\_distance+b\*reward\_hp+c\*reward\_distance\_2\* reward\_hp（其中a，b，c为正常数），并且满足当血量在一定阈值之后reward\_hp小于某个常数，理由如下（为方便简写为rd、rd\_2和rh，rd\_2的解释在下一段）：应为根据定义我们的rp非正，当拉近距离时rd为正，拉开距离时rd为负，那么我们希望满足的条件是a\*rd+b\*rh+c\* rd\_2\*rh<a\*(-rd)+b\*rh+c\*(-rd\_2)\*rh，最终可以推导得出rh<-a/c，即当血量满足一定阈值之后，此时再掉血得到的reward要小于常数-a/c才可以使agent此时有逃离的动作（在条件中我们假定拉近和拉开的距离近似相等，敌方使我方掉血近似相等做简化）。所以以上应该是我们最终的reward函数和满足的关系，并且每个reward函数具备的特有条件。

关于上式距离和血量的总reward=a\*reward\_distance+b\*reward\_hp+c\*reward\_distance\_2\* reward\_hp中的reward\_distance\_2函数应该大体和reward\_distance定义类似，但是对于到攻击范围5中不规范化为0，因为我们可以做如下考虑，如果reward\_distance\_2和reward\_distance函数一致，那么对于血量降到我们指定阈值后，如果agent在敌人攻击范围内时，因为步长的限制可能一步无法跨出攻击范围，那此时reward\_distance根据定义一定为0，这会导致上式对于在敌人攻击范围内掉血后的移动得到的reward和reward\_distance无关，这和我们的设计理念相违背，所以reward\_distance\_2去掉规范化后，对于在范围内血量到达阈值后掉血并拉近距离的reward会小于掉血并拉开距离的reward，这样符合我们的构思。

目前暂时写好了有关距离的reward\_distance函数，其为agent.py中的changeDistanceMatrix函数如下：

# reward\_distance

*def* changeDistanceMatrix(self,*obs*,*reaper\_i*):  
 distanceChanged=self.distance[*reaper\_i*]  
 self.distance[*reaper\_i*]=0  
 selected = *obs*.observation["screen"][\_SELECTED]  
 selected\_y, selected\_x = (selected == 1).nonzero()  
 player\_relative = *obs*.observation["screen"][\_PLAYER\_RELATIVE]  
 army\_y, army\_x = (player\_relative == \_PLAYER\_HOSTILE).nonzero()  
 *for* i *in* range(len(army\_x)):  
 y=abs(selected\_y[0] - army\_y[i])  
 x=abs(selected\_x[0] - army\_x[i])  
 *if* x\*x+y\*y<=25:  
 self.distance[*reaper\_i*] += 0  
 *else*:  
 self.distance[*reaper\_i*] += (x\*x+y\*y)\*\*0.5-5  
 distanceChanged=self.distance[*reaper\_i*]-distanceChanged  
 *return* distanceChanged

对于hp的reward函数和综合reward函数因为对于距离的reward函数的接入出现了许多bug，所以只是有了以上理论推导，但是还没有写到代码之中。

**1.2****代码改进：**

**1.2.1** **动作空间改进：**

原始代码中，动作空间只有3个——即等待，攻击，使用技能，这样对于我们的训练太过于简单，而且动作空间中没有移动对于我们最终希望达成的训练成果距离太远，所以我在动作空间中添加了上下左右四个移动的离散动作扩大了动作空间，并且上下左右移动的不长可以在combined\_action.py中自行设定，以向上移动为例，代码如下：

\_UP\_STRIDE=1

*def* moveUp(*obs*,*env*,*stride*=\_UP\_STRIDE):  
 selected = *obs*.observation["screen"][\_SELECTED]  
 selected\_y,selected\_x=(selected == 1).nonzero()  
 *if not* selected\_y.any():  
 action = [actions.FunctionCall(\_NO\_OP, [])]  
 obs = *env*.step(action)[0]  
 *if obs*.last():  
 *return False  
 else*:  
 *return obs  
 elif*(selected\_y[0]<40-*stride*):  
 target = [selected\_x[0], selected\_y[0]+*stride*]  
 action = [actions.FunctionCall(\_MOVE\_SCREEN, [\_NOT\_QUEUED, target])]  
 obs = *env*.step(action)[0]  
 *if obs*.last():  
 *return False  
 else*:  
 action = [actions.FunctionCall(\_NO\_OP, [])]  
 obs = *env*.step(action)[0]  
 *if obs*.last():  
 *return False  
 else*:  
 *return obs  
 return obs*

并且需要注意的是要将神经网络的初始输入改为7而不是原来的3才可以训练成功。

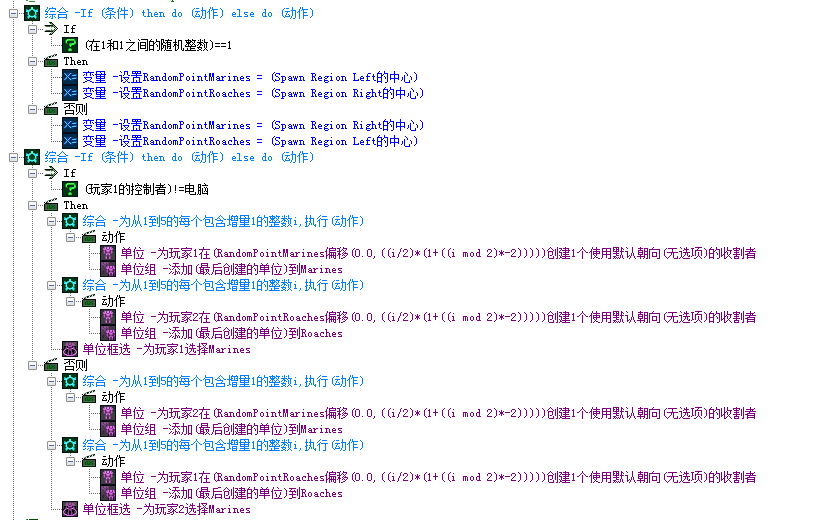
**1.2.2** **初始动作的改进：**

在原始代码中，因为没有移动，所以最开始的设定就是所有agent直接去攻击指定敌人，但是在移动空间中添加了移动后，我们需要修改初始直接攻击敌人，因为这样会影响我们的训练。所以在agent.py的build\_group函数中将相关代码暂时注释掉，最后全部修改完成后可以删除，具体可见修改后代码。

**1.3****地图条件改进：**

**1.3.1** **地图设置改进：**

在最初训练时就发现原始地图的初态是在小范围内随机生成的，但是对于maddpg此类与Q学习有关的方法，最好应该是对于确定初态的训练才能达到最好的效果，对此我修改了sc2地图的初始设定，新建了一个确定初态的地图no\_loop\_reapers\_2并进行了实验比较。地图设置的主要修改如图：



需要注意的是在修改地图的时候需要在地图编辑器中对于Init和Reset Map中都要修改才可以。

**1.3.2** **不同阵型改进：**

第三周的时候因为感觉原始阵型胜率十分之高，甚至高达90%，于是考虑对于其他阵型下测试maddpg算法，后来发现90%只是这个阵型的特例，对于其他阵型胜率基本都达不到这么高，所以对于不同算法、阵型、初态花费大量时间进行了对比测试（见1.4），对于不同初始阵型的设置只需要修改地图文件中的no\_loop\_reapers（随机初态）和no\_loop\_reapers\_2（确定初态）两个地图即可达到修改的效果。

**1.4****不同算法、阵型、初态对比测试：**

原始阵型：



胜率：

|  |  |  |
| --- | --- | --- |
|  | 随机初态（最原始方案） | 确定初态 |
| 原始maddpg算法 | 90.7% | 93% |
| 改进maddpg算法 | 90.5% | 95.3% |

修改阵型：



胜率：

|  |  |  |
| --- | --- | --- |
|  | 随机初态 | 确定初态 |
| 原始maddpg算法 | 42.9% | 68.0% |
| 改进maddpg算法 | 50.6% | 69.3% |

从以上实验可以看到不同阵型对于胜率影响非常大，而且随机初态对于强化学习为基础的方法影响很大，并且经过对比可以发现改进maddpg算法基本上可以说更胜原始算法一筹。

但是以上测试都在尚未修改动作空间时进行，当扩展动作空间后，胜率大幅度下降，对于原最好阵型当扩展动作空间后原始maddpg算法胜率只有15%左右，对于随机策略有20%左右，对于改进maddpg算法有35%左右。当去除最开始直接索敌的条件后胜率会进一步下降，这说明reward函数修改的必要性。

**2.****遗留问题：**

**2.1 控制帧的问题：**

首先简单描述现有控制逻辑，为了方便叙述假设我们有3个agent——A、B、C。对于从策略函数（μ函数）获得动作后，帧变化如下：

选择A→对A下达动作指令→选择A（获取现有观察值）→选择B（获取现有观察值）→选择C（获取现有观察值并将现有reward算为A执行动作后的reward）→选择B→对B下达动作指令→选择A（获取现有观察值）→选择B（获取现有观察值）→选择C（选择现有观察值并将现有reward算为B执行动作后的reward）→选择C→对C下达动作指令→选择A（获取现有观察值）→选择B（获取现有观察值）→选择C（选择现有观察值并将现有reward算为B执行动作后的reward）→将最后三帧获得的观察值算作A、B、C执行动作后的观察值（这并非一帧，但是要列出）

然后再从策略函数获取动作进行循环，这样会导致一个显而易见的问题，即观察值的不准确，并且当agent越多时这个问题是越严重的。因为原始maddpg算法中的基本设想是所有agent的动作应该算作同时执行，然后再在执行结束后同时执行观察获取观察值，现在的控制逻辑显然是违反我们的初衷的。

并且此时还有一个reward问题，如果我们对A下命令去攻击距离较远的敌人D，对B下达原地等待命令，那么此时会出现reward冲突的问题，因为计算A的reward的时候，可能因为距离问题A还未攻击到敌人，此时的reward只有一个距离reward。当B等待并计算reward的时候，如果此时A正好可以攻击到D并击杀后，这个reward如果没有特殊计算，那么就会盲目的加给B，这就会出现reward错误的问题。有关杀敌的reward问题主要可以归结为reward延迟问题。这种延迟会导致杀敌reward在某些情况下无法回馈给相应的动作从而导致错误的价值函数（Q函数）估计

**2.2 奖励函数（reward函数）的问题：**

主要reward函数的设想已经在1.1.3进行了详细的讨论，但是对于已经写好有关reward\_distance函数因为原有对于agent控制的逻辑问题还未完全接入，需要继续修改已有代码。

另一个关键点是对于敌人死亡的reward，应该更进一步细化到敌人掉血就获得reward才应该更好，而非现在的杀死敌人后才获得reward。但是对于reward还存有一个问题就是即时获取问题，这涉及到2.2中提及的控制帧的问题。

**3.****针对遗留问题的设想方案：**

对于在2中提出的两个问题首先考虑控制帧的问题，我们可以做如下修改，选择A→对A下达动作指令→选择B→对B下达动作指令→选择C→对C下达动作指令→选择A（获取现有观察值并将现有reward算为A执行动作后的reward）→选择B（获取现有观察值并将现有reward算为B执行动作后的reward）→选择C（选择现有观察值并将现有reward算为C执行动作后的reward）→将最后三帧获得的观察值算作A、B、C执行动作后的观察值（这并非一帧，但是要列出），这样的话我们的观察值可以尽量的准确，但同时也带来了一个问题就是reward会出现一定的延迟，不过对于较少的agent，如现有5个agent其带来的延迟问题尚可以接受。但是同样无法解决的问题就是也会出现对于杀敌奖励错乱的问题。这样我们在可以接受的reward延迟内解决了观察值不准的问题。但是对于控制帧的修改会导致现有逻辑出现很多问题，尤其是在判定一个控制单位是否死亡的时候，所以需要大量修改代码才可以。

对于攻击reward问题我个人认为应该是细化到地方掉血就获得reward，但是这样需要考虑掉血的来源，这是一个难题。并且对于因为攻击存在范围导致攻击指令所带来的reward延迟问题暂时没有一个好的想法来解决。