pysc2-rl-agents

唐浩然 1800012976

马康淇 1800012973

李心怡 1700015439

实验环境

tensorflow == 1.15.0

SC2：Linux SC2 4.0.2

Windows: Battle.net 最新版（5.0.4）

项目源代码地址：[CindyLi-Ayi/pysc2-rl-agents (github.com)]

项目分工：本小组由于任务多为大家一起肝出来的，所以一致决定任务量均匀分配！！！

Table of Contents

pysc2-rl-agents
- Table of Contents
- Code
- Run
- Base model
  - Input_preprocess
  - Network
  - A2CAgent
  - Main
- Result for each mini-game
  - MoveToBeacon
  - CollectMineralShards
  - FindAndDefeatZerglings
  - DefeatRoaches
  - DefeatZerglingsAndBanelings
  - Collect Minerals and Gas
  - BuildMarines

Code

我们的rl-based 模型都已游戏名的形式放在根目录的文件夹中；

rulebase 模型统一以游戏名称放在rule-base文件夹下；

rulebase的运行方法见rulebase文件夹中readme.txt

Run

- 如果对已有参数进行test，需要先解压snapshot文件夹下面的参数文件

```
> python -m main --map=MoveToBeacon --training=False
```

Base model

- 所有minigame都基于跟原始论文中全卷积网络相似的结构，但是用epsilon-greedy的方法进行探索。

Input_preprocess

- 对空间输入（minimap和screen中的每一层），我们对定量变量进行归一化，对分类变量进行one-hot encoding。其中有两个层的分类变量过多，因此也采取归一化的方法。

```
def preprocess_minimap(minimap):
  """
  one-hot encode categorical, normalize scalar/player_id inputs
  :return: c * 64 * 64 input layer
  """
  layers = []
  for i in range(len(features.MINIMAP_FEATURES)):
    ## scalar or to large to do one-hot
    if i == _MINIMAP_PLAYER_ID or features.MINIMAP_FEATURES[i].type ==
features.FeatureType.SCALAR:
      layers.append(minimap[i:i+1] / features.MINIMAP_FEATURES[i].scale)
    ## categorical
    else:
      layer = np.zeros([features.MINIMAP_FEATURES[i].scale,
                        minimap.shape[1],
                        minimap.shape[2]],
                       dtype=np.float32)
      for j in range(features.MINIMAP_FEATURES[i].scale):
        indy, indx = (minimap[i] == j).nonzero()
        layer[j, indy, indx] = 1
      layers.append(layer)
  return np.concatenate(layers, axis=0)
```

- 对于非空间的信息，我们在代码中留了接口，但是后续没有使用。大部分都是用rule调用。

Network

- 网络结构与论文中的FCN基本一致，但是缩小了空间信息提取时的channel个数（缩小为原先的一半）。
  - minimap 和 screen 中的空间信息分别通过两个卷积层
  - 非空间信息通过一个全连接层，和tanh激活层
  - 提取后的空间信息channel叠在一起， 再经过一个channel=1的卷积层，最后通过softmax层输出空间参数
  - 提取后的空间和非空间信息横向拼接在一起，再经过一个全连接层，和relu激活层，形成feature层
  - feature层通过一个全连接层，再通过一个softmax层输出非空间动作
  - feature层通过另外一个全连接层，输出Q value

```
def build_net(minimap, screen, structure, num_action):
  """
```

```
    same network structure as Fully Convolution Network in paper
    :return: spatial_action, non_spatial_action, Q-value
    """
    ## feature extraction
    minimap_conv1 = layers.conv2d(tf.transpose(minimap, [0, 2, 3, 1]),
                          num_outputs=4,
                          kernel_size=5,
                          stride=1,
                          scope='minimap_conv1')
    minimap_conv2 = layers.conv2d(minimap_conv1,
                          num_outputs=8,
                          kernel_size=3,
                          stride=1,
                          scope='minimap_conv2')
    screen_conv1 = layers.conv2d(tf.transpose(screen, [0, 2, 3, 1]),
                          num_outputs=4,
                          kernel_size=5,
                          stride=1,
                          scope='screen_conv1')
    screen_conv2 = layers.conv2d(screen_conv1,
                          num_outputs=8,
                          kernel_size=3,
                          stride=1,
                          scope='screen_conv2')
    structure_fc = layers.fully_connected(layers.flatten(structure),
                                    num_outputs=64,
                                    activation_fn=tf.tanh,
                                    scope='structure_fc')

    ## spatial action branch
    feature_conv = tf.concat([minimap_conv2, screen_conv2], axis=3)
    spatial_action = layers.conv2d(feature_conv,
                                  num_outputs=1,
                                  kernel_size=1,
                                  stride=1,
                                  activation_fn=None,
                                  scope='spatial_action')
    spatial_action = tf.nn.softmax(layers.flatten(spatial_action))

    ## non spatial actions & Q-value branch
    feature_fc = tf.concat([layers.flatten(minimap_conv2),
layers.flatten(screen_conv2), structure_fc], axis=1)
    feature_fc = layers.fully_connected(feature_fc,
                                    num_outputs=256,
                                    activation_fn=tf.nn.relu,
                                    scope='feature_fc')
    non_spatial_action = layers.fully_connected(feature_fc,
                                                num_outputs=num_action,
                                                activation_fn=tf.nn.softmax,
                                                scope='non_spatial_action')
    value = tf.reshape(layers.fully_connected(feature_fc,
                                              num_outputs=1,
                                              activation_fn=None,
                                              scope='value'), [-1])

    return spatial_action, non_spatial_action, value
```

A2CAgent

- 在 `step` 函数里定义了agent如何从obs输入计算出动作，下面是step函数的一部分
  - 对于网络的输出 `spatial_action`，选取其最大值
  - 对于网络的输出 `non_spatial_action`，选取当前允许选取的动作当中的最大值
  - 如果该动作需要非空间参数，假设为0。大部分情况下都是最优选择
    - 例如，queue参数为0时选取的是'now'
  - 在 `epsilon[0]` 的概率下选择随机动作，在 `epsilon[1]` 的概率下选择随机空间参数

```
## choose spatial and non-spatial action
non_spatial_action = non_spatial_action.ravel()
valid_actions = obs.observation['available_actions']
act_id = valid_actions[np.argmax(non_spatial_action[valid_actions])]

spatial_action = spatial_action.ravel()
target = np.argmax(spatial_action)
target = [int(target // self.ssize), int(target % self.ssize)]

## epsilon greedy exploration
if self.training and np.random.rand() < self.epsilon[0]:
  act_id = np.random.choice(valid_actions)
if self.training and np.random.rand() < self.epsilon[1]:
  range = int(self.random_range)
  dy = np.random.randint(-range, range)
  target[0] = int(max(0, min(self.ssize-1, target[0]+dy)))
  dx = np.random.randint(-range, range)
  target[1] = int(max(0, min(self.ssize-1, target[1]+dx)))

## return function
act_args = []
for arg in actions.FUNCTIONS[act_id].args:
  if arg.name in ('screen', 'minimap', 'screen2'): ## spatial arg
    act_args.append([target[1], target[0]])
  else:
    act_args.append([0])  ## non-spatial arg

return actions.FunctionCall(act_id, act_args)
```

- 在 `build_model` 里规定了计算loss和更新参数的流程
  - 从目标Q value和网络Q value上计算出advantage
  - 分别计算空间动作的概率和非空间动作在能够采取的非空间动作中的softmax概率，乘以advantage计算出policy loss
  - 网络Q value乘以advantage计算出value loss
  - 二者相加用RMSProp更新参数

```
## compute log probability
spatial_action_prob = tf.reduce_sum(self.spatial_action *
self.spatial_action_selected, axis=1)
spatial_action_log_prob = tf.log(tf.clip_by_value(spatial_action_prob, 1e-
10, 1.))
```

```
non_spatial_action_prob = tf.reduce_sum(self.non_spatial_action *
self.non_spatial_action_selected, axis=1)
valid_non_spatial_action_prob = tf.reduce_sum(self.non_spatial_action *
self.valid_non_spatial_action, axis=1)
valid_non_spatial_action_prob =
tf.clip_by_value(valid_non_spatial_action_prob, 1e-10, 1.)
non_spatial_action_prob = non_spatial_action_prob /
valid_non_spatial_action_prob
non_spatial_action_log_prob =
tf.log(tf.clip_by_value(non_spatial_action_prob, 1e-10, 1.))

## policy loss & value loss
action_log_prob = self.valid_spatial_action * spatial_action_log_prob +
non_spatial_action_log_prob
advantage = tf.stop_gradient(self.value_target - self.value)
policy_loss = - tf.reduce_mean(action_log_prob * advantage)
value_loss = - tf.reduce_mean(self.value * advantage)
self.summary.append(tf.summary.scalar('policy_loss', policy_loss))
self.summary.append(tf.summary.scalar('value_loss', value_loss))
loss = policy_loss + value_loss

## RMSProp optimizer
self.learning_rate = tf.placeholder(tf.float32, None, name='learning_rate')
opt = tf.train.RMSPropOptimizer(self.learning_rate, decay=0.99, epsilon=1e-
10)
grads = opt.compute_gradients(loss)
cliped_grad = []
for grad, var in grads:
  self.summary.append(tf.summary.histogram(var.op.name, var))
  self.summary.append(tf.summary.histogram(var.op.name+'/grad', grad))
  grad = tf.clip_by_norm(grad, 10.0)
  cliped_grad.append([grad, var])
self.train_op = opt.apply_gradients(cliped_grad)
self.summary_op = tf.summary.merge(self.summary)

self.saver = tf.train.Saver(max_to_keep=100)
```

- 学习率以如下方式decay

```
self.lr = self.original_lr * (1 - 0.9 * self.cur_episode / self.max_episode)
```

Main

- 包括了一些默认参数

```
flags.DEFINE_float("learning_rate", 5e-3, "Learning rate for training.")
flags.DEFINE_float("discount", 0.99, "Discount rate for future rewards.")
flags.DEFINE_integer("max_episode", int(1000), "Total steps for training.")
flags.DEFINE_integer("snapshot_episode", int(100), "Step for snapshot.")
flags.DEFINE_float("epsilon_non_spatial", 0.05, "epsilon of choosing random
action")
flags.DEFINE_float("epsilon_spatial", 0.2, "epsilon of choosing random xy
location")
flags.DEFINE_integer("random_range", 5, "change of xy location")
flags.DEFINE_integer("mean_episodes", 100, "How many episode for mean")
```

Result for each mini-game

MoveToBeacon

baseline = 25

best_mean_score = 26

max_score = 35

思路：起初的想法是用我们的A2C网络不加修改，希望其可以学习出可行的spatial动作+操作信息和non_spatial动作，直接观察训练结果，在gpu上训练10k轮左右，发现了如下问题。

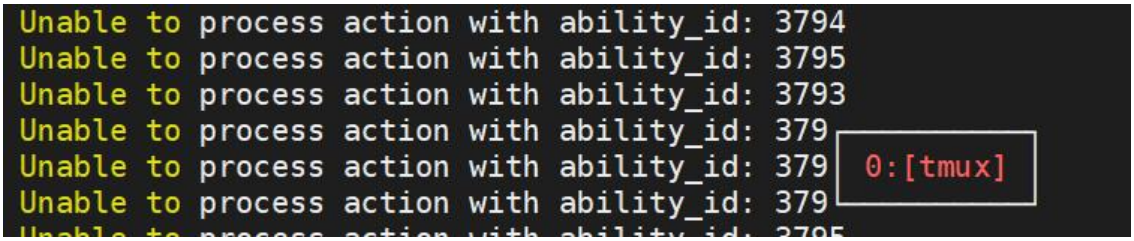
1. 输入feature_layers过多，其中能利用的信息极少
2. 网络过大，训练速度缓慢
3. 训练结果会发生突变，在2k轮左右会突然结果变好

在此基础上我们人工对于网络进行了修改，减少了卷积网络的卷积核，同时减少了全连接网络的节点数；同时，对于输入input_preprocess进行了人为规定：

```
_MINIMAP_SELECTED = features.MINIMAP_FEATURES.selected.index
_MINIMAP_PLAYER_RELATIVE = features.MINIMAP_FEATURES.player_relative.index
_SCREEN_PLAYER_RELATIVE = features.SCREEN_FEATURES.player_relative.index
_SCREEN_UNIT_TYPE = features.SCREEN_FEATURES.unit_type.index
_SCREEN_SELECTED = features.SCREEN_FEATURES.selected.index
```

只选取了如下五个feature_layer进行输入，训练效果有所改进，但还是存在如下问题。

1. 初始网络的随机情况会严重影响训练结果，我们发现如果agent在前期找不到beacon，后续会导致长时间得分都为0，而得分为0会导致agent长时间学不到东西。
2. MoveToBeacon这个游戏的reward极其稀疏，前期上千步只能得到得到1的奖励，人工会看发现可能agent并不能区分出具体是哪一种操作才是得到reward的原因。
3. Linux上SC2环境与windows上的差别，我们选用的是SC24.0.2的Linux版本，而windows上是战网最新版的sc2，发现Linux经常会出现如下的报错而Windows上不会。查询动作空间之后发现都是与屏幕移动相关指令如331（Move_Screen）




```
Unable to process action with ability_id: 3795 (3794,3793)
```

实验中发现，agent经常会走到地图的某一个角落然后一直往角落中走，怀疑可能是由于坐标越界，但是打印出结果后发现坐标并没有越界，在调试后也没有找到直接原因，故认为其可能是版本的bug。

在此基础上又做了如下的人工修改：

在action的选取上对其进行了规定，使用12（attack）作为固定动作，原模型只需学出选取的坐标即可，选择12的原因是331在linux上可能会出现问题，而attack指令不会出现类似的bug。

```
mask_id = 7
    for i in valid_actions:
      if i == 12:
        mask_id = 12
    act_id = mask_id
```

同时对于reward也进行了如下修改，对于在后面的beacon会给一个额外的奖励，以期待其可以找到尽可能高分的策略：

```
reward = obs.reward * (obs.observation["score_cumulative"][0] + 1) * 10
```

在这样的前提下只需训练1k轮左右就可以得到一个比较好的模型，但是结果仍然对初值有一定的依赖性，所以我们选取了其中表现较好的模型。

CollectMineralShards

baseline = 96

best_mean_score = 182 (rule based)

best_mean_score = 71 (Our rl model)

思路：与MoveToBeacon类似，直接使用全部的feature信息对于训练来说是很困难的，在前几千轮训练中并不能获得什么有意义的结果，因此在RL模型中采用了同样的方法缩减了输入的特征空间并且对可选的动作进行了限制。同第一个任务，在Linux服务器上进行训练会出现Move_screen函数出错的bug，而在本地windows系统运行则不会，因此在限制动作中删除了Move_screen，以Attack_screen动作作为移动方式。输入特征仍然为：

```
_MINIMAP_SELECTED = features.MINIMAP_FEATURES.selected.index
_MINIMAP_PLAYER_RELATIVE = features.MINIMAP_FEATURES.player_relative.index
_SCREEN_PLAYER_RELATIVE = features.SCREEN_FEATURES.player_relative.index
_SCREEN_UNIT_TYPE = features.SCREEN_FEATURES.unit_type.index
_SCREEN_SELECTED = features.SCREEN_FEATURES.selected.index
```

该任务的优势是Shards的位置在全图分布较多，因此随机移动也很容易获得reward，因此RL方法有很大希望学到合理的policy。在实验中，经过约700轮的训练，agent的得分从平均在10左右暴涨到了60左右，又经过一段时间训练平均大约在70左右。在测试过程中观察到，model学到了将两个marine同时选中并遍历地图上的Shards，但是这种结果没达到baseline，分析有2个原因：

1.模型没能学会将两个marine分别控制采集，这样会大幅提升采集速度，然而交替操作两个marine是很困难的，大部分情况下对两个marine分别控制结果反而不如一起控制，因此模型可能会认为这种策略是不好的，因而很少进行这种策略的探索。

2.由于不同操作系统之间的bug我们要用attack_screen代替move，但是模型采用的Attack_screen函数进行移动不允许直接点在shard上（因为shard不可攻击），所以需要学到点在shard的周围，增大了学习难度。同时attack动作可能会攻击同伴引起自杀，增大了负收益的风险。

在分析到这两个困难之后，我们在服务器上继续运行了3000轮训练，最终得分基本没有变动了。因此这个model在短时间内不太可能探索出分兵方式（两个marine一起移动即便严格采用了贪心方法，最高的平均分也仅为60-80之间），因此我们为了解决这个问题又做出了一个rule base的agent：策略是每次给出动作时，以所有Shard的坐标计算平均值，按左右两边分别让两个marine分别进行采集，采集顺序采用贪心法找最近的一个。最终得到的agent达到了同时采集的2倍。

FindAndDefeatZerglings

baseline = 44

best_mean_score = (rule based)

best_mean_score = 19 (Our rl model)

对于这个任务，我们在最初用原始模型训不出结果后沿用了MoveToBeacon的设计思路：

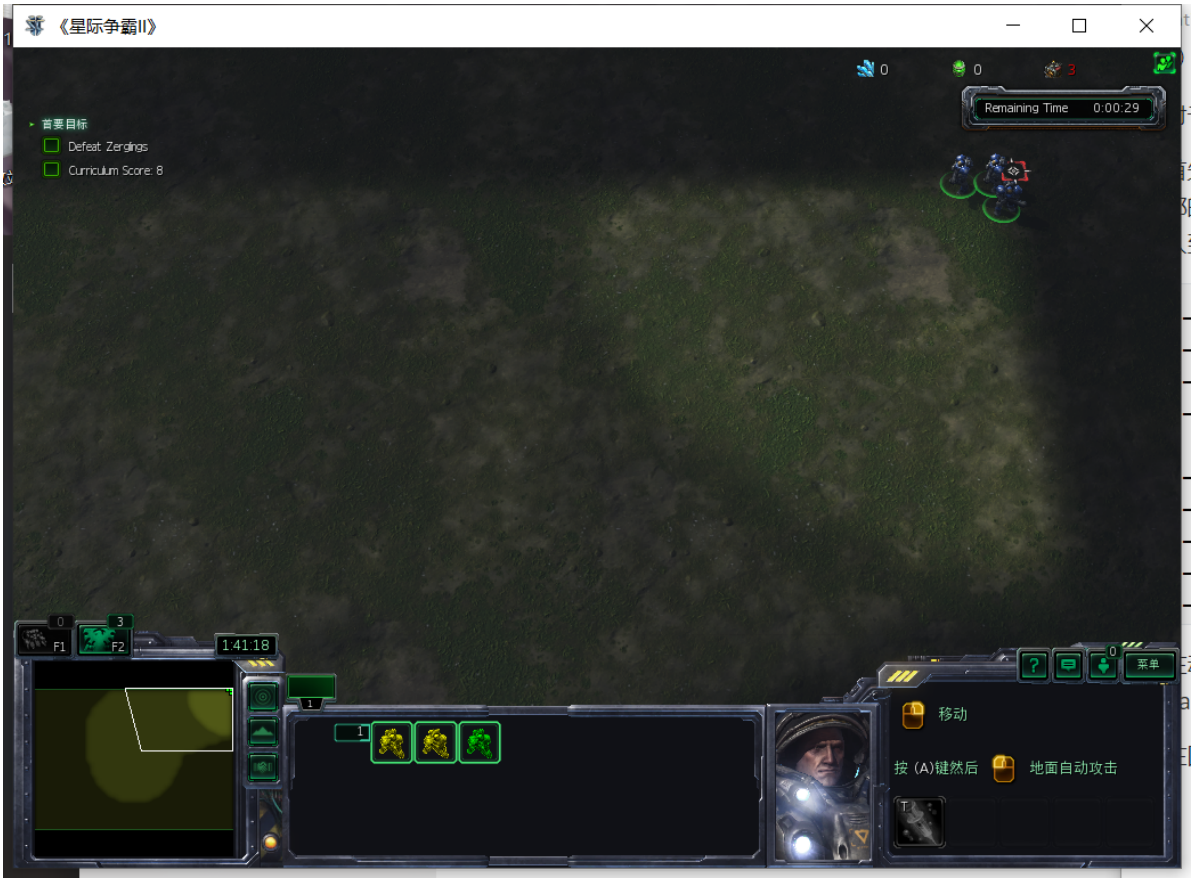
首先这个任务包含战争迷雾，也就是可视区域有限，这个游戏最特殊之处就在于screen里面并没有包含全部的信息，所以在feature_layer的选取上做出了修改，由于这个问题士兵有hit_point信息，所以也将其加入到输入中：

```
_MINIMAP_SELECTED = features.MINIMAP_FEATURES.selected.index
_MINIMAP_PLAYER_RELATIVE = features.MINIMAP_FEATURES.player_relative.index
_MINIMAP_VISIBILITY = features.MINIMAP_FEATURES.visibility_map.index
_MINIMAP_CAMERA = features.MINIMAP_FEATURES.camera.index

_SCREEN_PLAYER_RELATIVE = features.SCREEN_FEATURES.player_relative.index
_SCREEN_UNIT_TYPE = features.SCREEN_FEATURES.unit_type.index
_SCREEN_SELECTED = features.SCREEN_FEATURES.selected.index
_SCREEN_VISIBILITY = features.SCREEN_FEATURES.visibility_map.index
_SCREEN_HIT_POINTS = features.SCREEN_FEATURES.unit_hit_points.index
```

在动作的选取上，想到既然screen信息不全不如直接选取位于minimap上的操作，于是选择patrol_minimap（334）作为输出动作的选择：

在回放训练过程时发现在对于地图的探索上，我们的agent会优先选取一个方向进行探索，左上或右上

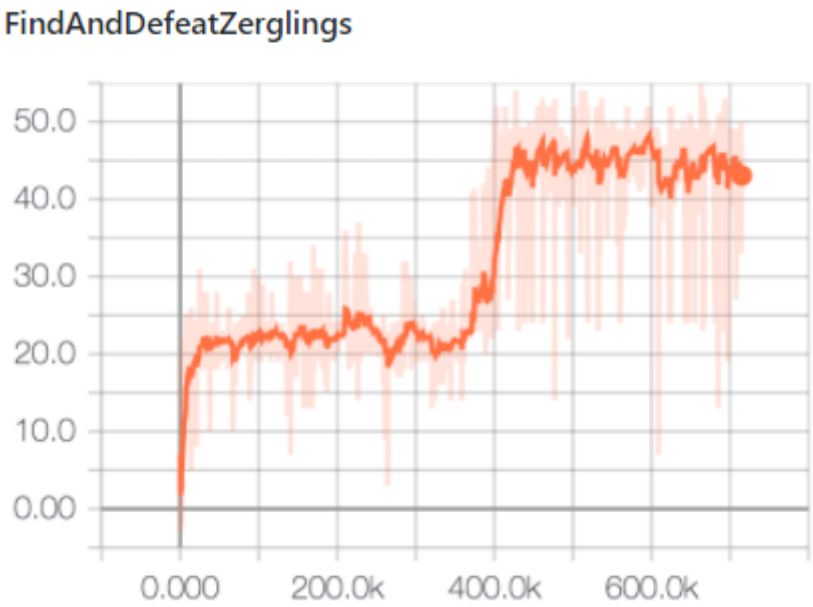


之后会在很长一段时间都卡在这里，得分一直是在5-8之间。

于是考虑修改了reward，对于没有reward返回的情况会返回-1，即避免其长时间没有探索。在经过更多轮训练后发现其逐渐学会探索其他方向，并在少数情况出现了转一圈的情况。（下图为探索了两个方向）

但在windows上跑了接近5k轮次之后发现得分仍然上不去20，也就是没有进入到第二轮的reward中，考虑原因应该是训练量不够，通过人工看回放分析认为我们的模型是可以学到有效策略的。但是由于时间原因在最后这个任务我们小组选择了rule-base的方法。

于此同时，在这个任务上，我们小组在Linux的服务器上面用我们最初的模型进行了测试（也就是没有加人工的情况），将我们的模型进行了改进利用A3C的思想，开了四个线程对这个任务进行训练，在训练24h之后每个线程运行了约8k轮，最后得到的测试结果平均只有19分，部分情况可以达到20分以上。对此的解释我们认为依旧是训练时间不足，参考了github上其他人给出的训练结果（下图为github上的可视化[simonmeister/pysc2-rl-agents: StarCraft II / PySC2 Deep Reinforcement Learning Agents (A2C) (github.com)](simonmeister/pysc2-rl-agents: StarCraft II / PySC2 Deep Reinforcement Learning Agents (A2C) (github.com))



我们认为当前我们的训练还处在上图中间部分的平缓区域，在继续训练的过程中会有极大可能学会完成第二轮的任务，但这一步需要很长的训练时间。

最终给出了rule-base的方法：

DefeatRoaches

baseline = 98

best_mean_score = 48 （our rl model）

max_score = 315 (our rl model)

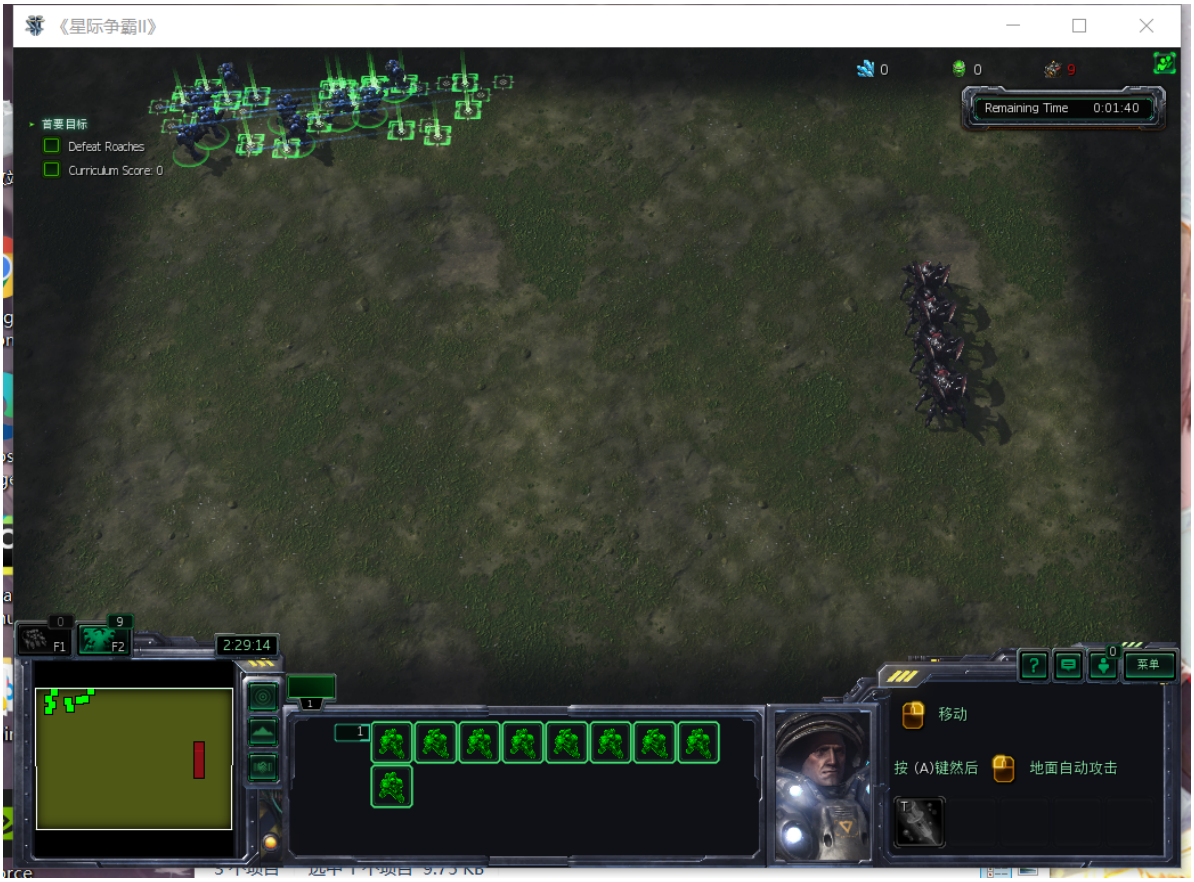
best_mean_score = （out rule-base model）

这个任务的设计上同样修改了输入、动作空间：

输入的layer与任务3相似，但是去掉了visibility和camera，因为并不会在实际中被应用。

动作空间上选取了screen_patrol（334）作为输出动作。

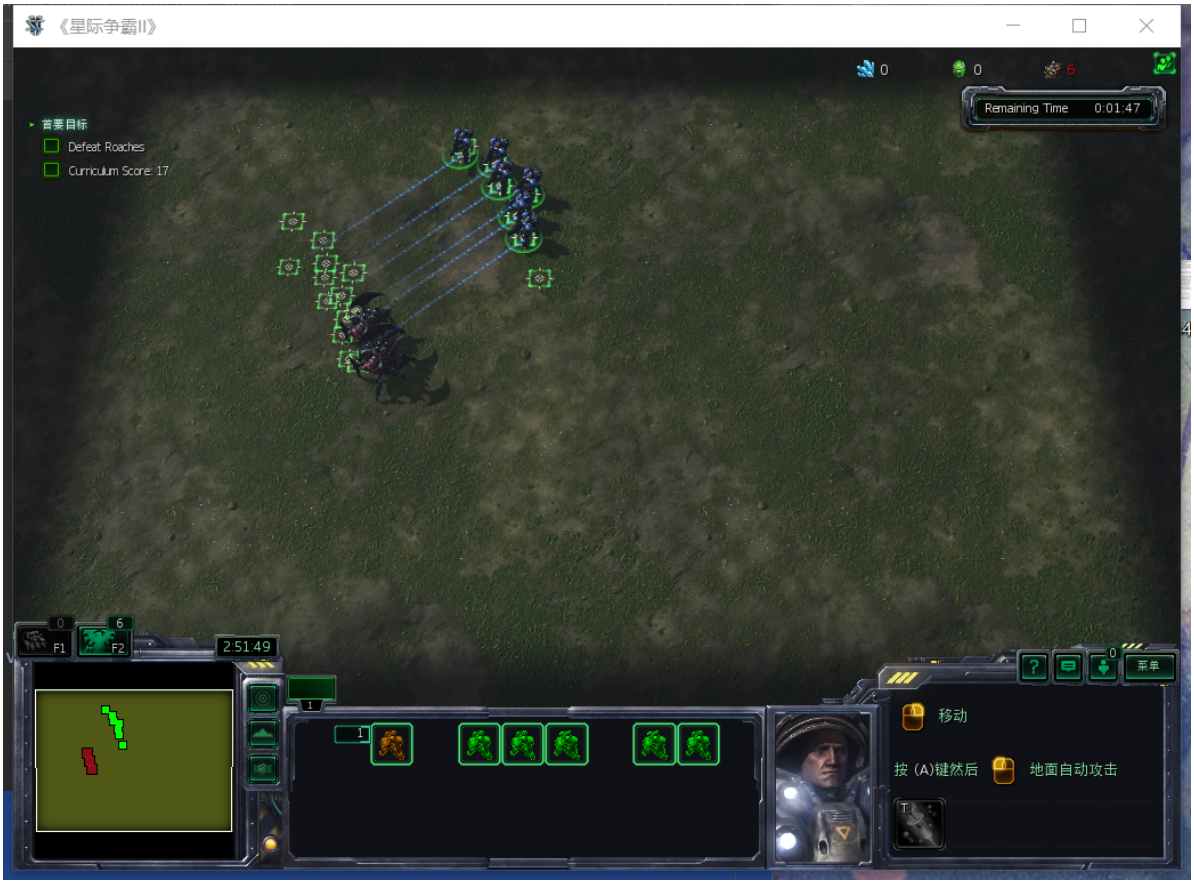
在训练过程中发现agent会有避战的操作：



这样会导致agent长时间并不能得到有效的学习，于是人为的修改了reward：

使其在没有战斗的情况下reward = -1（即没有出现两方伤亡的情况），以此来让agent不管如何都会主动迎战，（这部分考虑并不是很周到，reward的修改可能会带来其他负面影响，比如如果已经确认打不过而且会减分就不需要再去打，但是这部分影响不会成为主要因素）

在经历了1k轮左右的训练就学会了与对方战斗



中间一段时间甚至学会了从一侧集火对面单位，但在4k轮左右情况就变成了下图：

会发现其总是会趋向左右两个角落，并把自己排成接近一字型，探索一段时间后就又回到原来的正常作战之后会经历类似的循环，这部分的可解释性并不强，我们选取了其中表现相对好的时机作为checkpoint，平均水平就只有50左右，于是该任务也考虑了人工rule-base手段。

DefeatZerglingsAndBanelings

baseline = 62

best_mean_score = 52 (our rl model)

best_mean_score = (our rule-baesd model)

这个问题与任务四类型几乎一致，不同的是其中的兵种Banelings和Zerglings都是近战单位，所以学出的策略会有所差别，另一个有趣的差别就是两者敌军初始队形是不同的，这点对于我们的agent产生了一定的影响。

在这个任务上，相较于上一个任务，我们的模型表现更好，在平均上已经接近论文中的baseline，由于希望达到baseline的得分，我们后续又做了一个基于规则的模型。

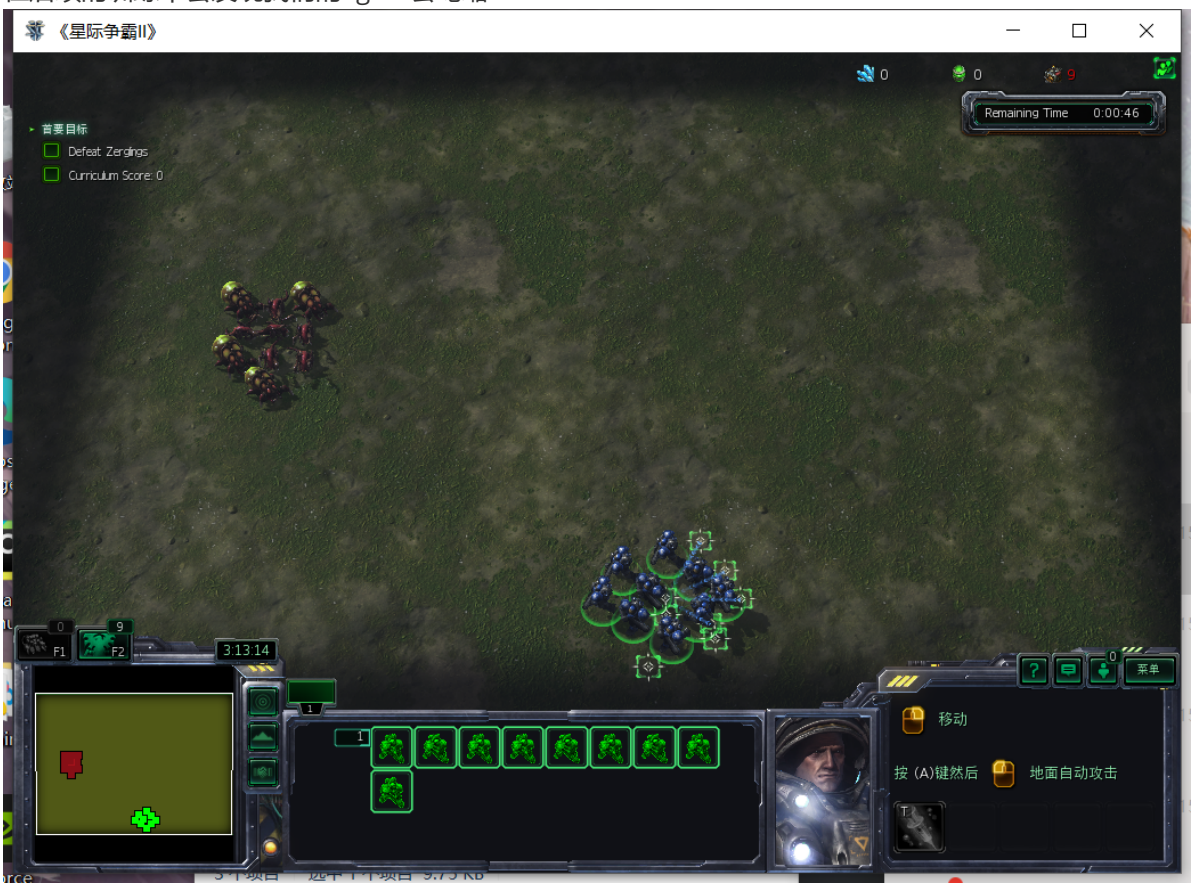
在本次rl模型的实现上，模型的搭建与任务4完全一致，在学习过程中我们发现了如下有趣的现象：

起初与上一个任务相似，也是直接一排过去与对方交战：

在后续训练中会出现蜷缩在一个角落避战的情形：

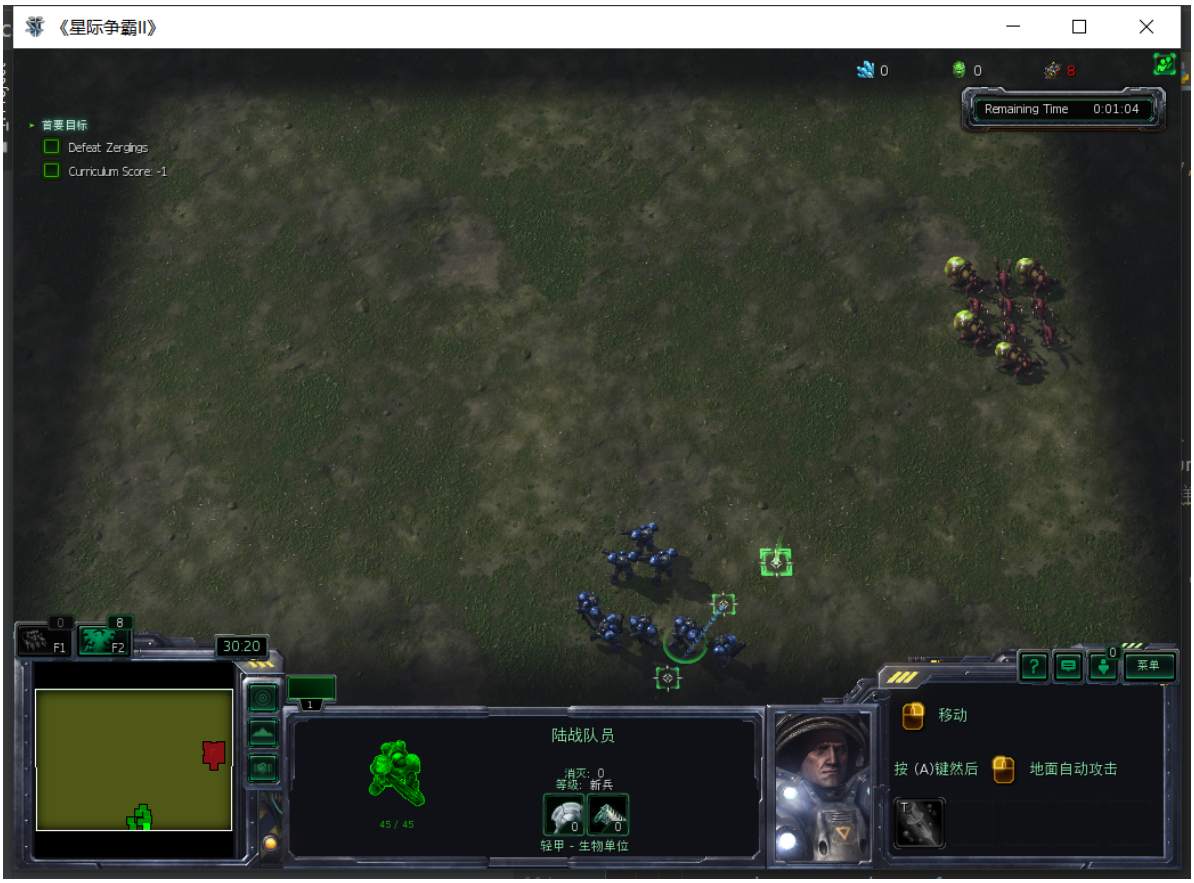
但在这里，我们发现，有趣的是部队的分布是不同的。在上一个任务中会趋向于排成一排，而在此次任务中是蜷缩成一团，目前我们的观点是其与初始怪物的站位是相关的。

在后续的训练中会发现我们的agent会蜷缩



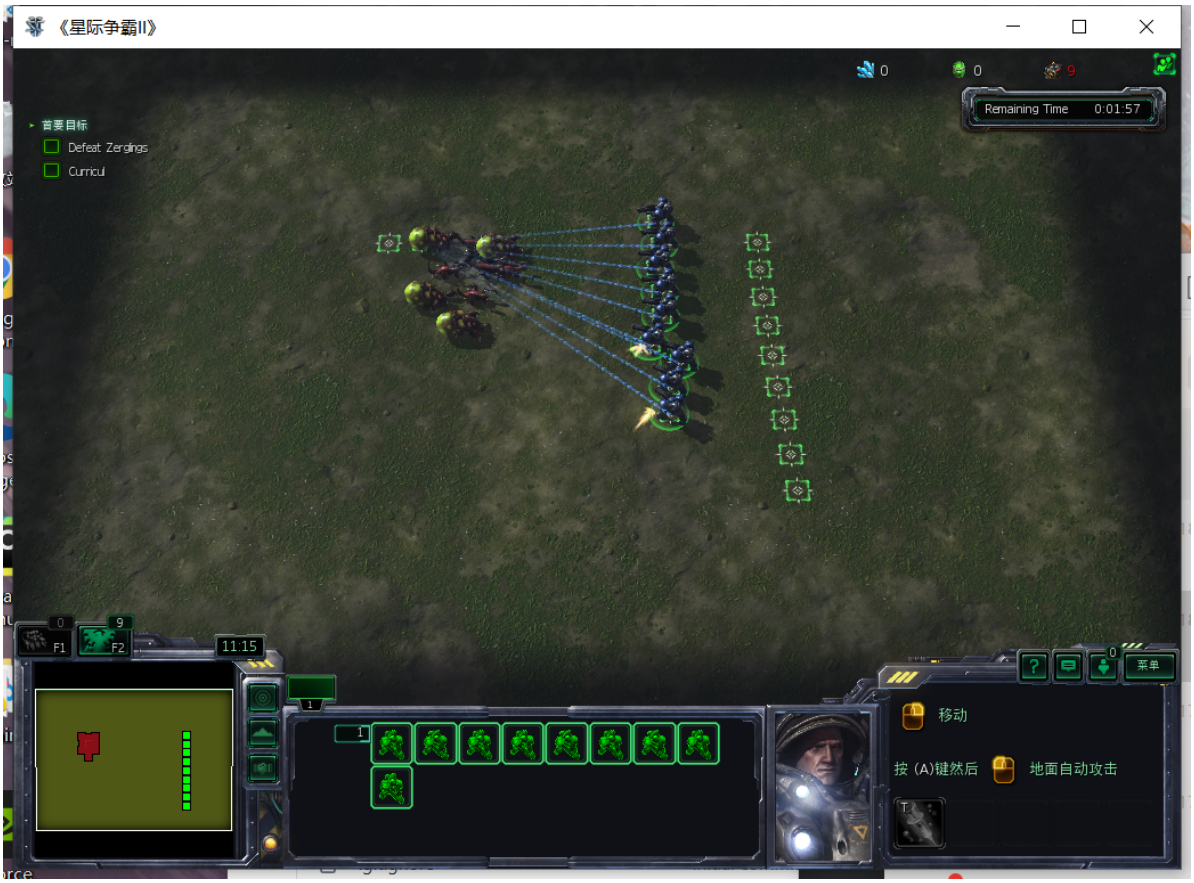
成一团，并与怪物保持一个极限的安全距离。

这样会导致长期得不到收益，于是后面会发现我们的agent竟然学会了分出一个兵去勾怪（也可能只是偶然发现）



这样会得到一定的收益，但得分始终上不去。

后续又经历了战斗和避战的不断循环，有一段时间学会了优先集火其中的自暴虫（Benalings），由于其冲过来会造成大量伤害。



由于我们的训练只有13k左右，所以效果上仍然起伏比较大，我们选取了一个相对稳定的checkpoint，得到的平均值在50左右，但是我们相信在给一定的训练时间该模型一定可以达到一个较为理想的结果。迫于时间不足，又急于过baseline，于是我们同样给出了rule-based的方法。

Collect Minerals and Gas

Baseline: 3351

Best score: 3925

Best mean: 3841

- 总共尝试了以下几个版本：

1. Base model
2. 限制输入空间到`screen_feature.relative`，动作空间到给定的5个有用动作（无动作、选中大本营、造scv、选中空闲scv、采矿）
3. 限制输入空间到`screen_feature.relative`，动作空间到给定的3个有用动作（无动作、选中空闲scv、采矿）
4. 限制输入空间到`screen_feature.relative`中的`NEUTRAL`层，动作空间中非空间动作选择给定（一旦可能就选中大本营并造scv，此外一旦有空闲scv就进行采矿），部分空间参数由网络自己选择（选中大本营空间参数给定，采矿空间参数）。rule部分的代码如下

```
if 490 in obs.observation.available_actions:
  # print('train scv')
  return actions.FunctionCall(490, [[0]])
```

```
if obs.observation['player'][3] < obs.observation['player'][4] and
obs.observation['player'][1] > 50:
  return actions.FunctionCall(2, [[0], [25, 25]])
if not self.walked and len(obs.observation['multi_select']) > 0 and 264 in
obs.observation.available_actions:
  # print('harvest')
  self.walked = True
  return actions.FunctionCall(264, [[0], target])
if 0 in obs.observation.available_actions and obs.observation['player'][-4]
== 0:
  # print('noop')
  return actions.FunctionCall(0, [])
if 6 in obs.observation.available_actions and obs.observation['player'][-4]
> 0:
  # print('select')
  self.walked = False
  return actions.FunctionCall(6, [[1]])

# print('noop')
return actions.FunctionCall(0, [])
```

- 此外，对网络进行了如下改进和调整：针对该任务，原始网络的对动作空间的探索不够，体现在epsilon和random_range过小。这是因为，该动作当中唯一会改变input layer的动作采矿，如果动作目标没有点击在矿上，scv根本不会动。这就意味着，如果一开始网络的输出即使加上探索也没有可能点击在矿上，网络的训练就无法进行，因此我们设计了递减的epsilon和random_range。

```
flags.DEFINE_float("epsilon_non_spatial", 1, "epsilon of choosing random
action") ## =0 for version 4
flags.DEFINE_float("epsilon_spatial", 1, "epsilon of choosing random xy
location")
flags.DEFINE_integer("random_range", 32, "change of xy location")

## 在reset函数中
if self.epsilon[1]>0.1:
  self.epsilon[1] *= 0.999
if self.random_range>1:
  self.random_range *= 0.999
```

- 其中，前三个版本均无法跑出有效结果，原因总结如下：
  - 当使用原网络时，算力不足，网络比原始paper里的网络小，算的轮数等等都不够
  - v2v3被寄予很大期望，但是仍然受限于“如果网络输出的空间参数不在矿上，那么就一直拿不到分“的困境。当epsilon逐渐减小的时候，就可以发现training的score越来越差。所以实质上网络是在靠随机某一次点到矿上来得分，而不是学到了哪一次是对的。
  - v4按道理来说应该能训练出来，但是我们的训练结果并不稳定。目前snapshot里保存了一份成功的参数。

BuildMarines

baseline = <1

best_mean_score = <1 (our rl model)

best_mean_score = 11 (our rule-baesd model)

类似之前的任务，这个任务选择传入的特征层是screen和minimap的unit_type, player_relative，selected等信息，加上info这个非空间信息。动作限制在如下分析的几个动作当中。

然而任务相对而言比较困难，造marine的过程要先采矿，再选择scv，再造补给站，在选择scv造barrack，再选择barrack造marrine。而整个持续动作空间和状态空间是过于巨大的，在服务器上进行了6h约2k轮的训练结果全为0，只能依赖随机探索获得reward（而这个概率太低太低），因此想要和Deepmind的600K训练次数结果相比仍然差得很远，在观察训练结果中发现，由于没有reward，模型甚至连选择scv并建造补给站这第一步都学不到，因而觉得这个任务的纯RL方法是不太现实的。最后使用了一个简单的rule based模型：首先等待mineral足够多，然后简单的选择scv、制造补给站、制造兵营、并制造marine，这种简单的方法可以获得11分。