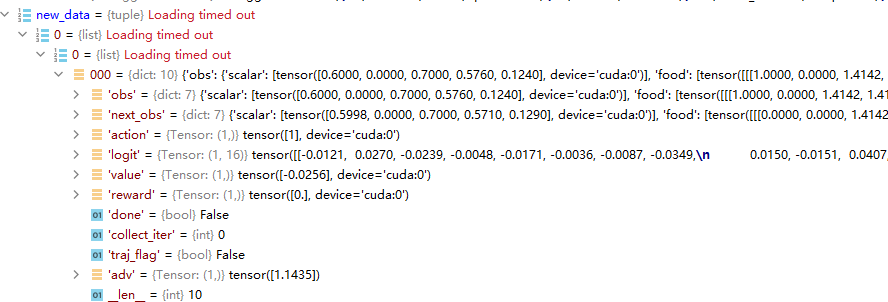
# PPO used in go-bigger

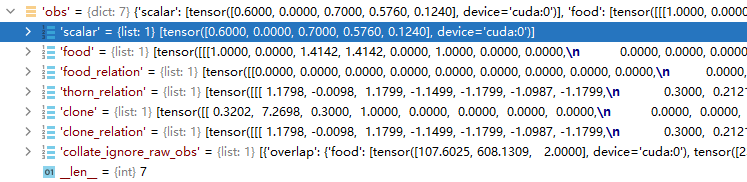
**一般DRL在实际使用时根据流程可分类为collect、learn、eval三大过程，所以本文将PPO代码分成这三大过程逐一解析，重点解析learn部分的原理与代码，最后边再附上共享参数的PPO网络模型以供参考**

## collect

同时输出 action 和 value， 并且保存在buffer中：

output = self.\_collect\_model.forward(data, mode='compute\_actor\_critic')

保存到 buffer 的数据信息结构如下，其中 **tarj\_flag**是一个标志位，从done信息得来但是将一组数据的最后一个done置为True：  


其中；obs下数据的格式为：  


每一次采集的数据可以多次输入到learn中进行学习，**提高样本的利用率**：

for i in range(cfg.policy.learn.update\_per\_collect):  
 input\_data = copy.deepcopy(train\_data)  
 learner.train(input\_data, collector.envstep)

此外，还使用了**并行采集**，使用了pipe库，具体使用方法还没有掌握

## learn

### 流程

（1）用data中的obs信息和next\_obs信息得出value和next\_value：并且对value信息进行标准化

with torch.no\_grad():  
 value = self.\_learn\_model.forward(data['obs'], mode='compute\_critic')['value']  
 next\_value = self.\_learn\_model.forward(data['next\_obs'], mode='compute\_critic')['value']  
 if self.\_value\_norm:  
 value \*= self.\_running\_mean\_std.std  
 next\_value \*= self.\_running\_mean\_std.std

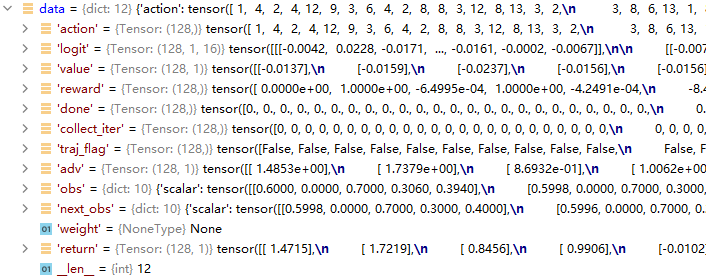
（2）利用data中的数据和value，next\_value得出adv信息：

compute\_adv\_data = gae\_data(value, next\_value, data['reward'], data['done'], data['traj\_flag'])  
data['adv'] = gae(compute\_adv\_data, self.\_gamma, self.\_gae\_lambda) # 0.9, 0.95

具体步骤在本小节后的关键步骤中。

（3）求return值，并对return和value进行标准化：

unnormalized\_returns = value + data['adv']  
if self.\_value\_norm:  
 data['value'] = value / self.\_running\_mean\_std.std  
 data['return'] = unnormalized\_returns / self.\_running\_mean\_std.std  
 self.\_running\_mean\_std.update(unnormalized\_returns.cpu().numpy())

处理后的data数据包含信息如图：  


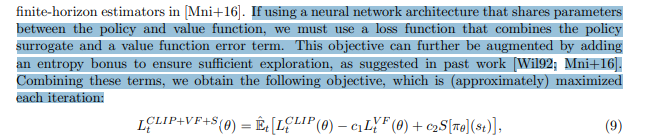
（4）将**data数据随机打乱**，分组，每组大小为learn\_batch，输入到model中进行loss计算，下面代码是计算loss时数据的准备工作（**这里要注意data的维度，因为数据的运算全都是并行batch形式，不注意检查很容易就会出现问题，之前在这里吃了大亏**）：

for batch in split\_data\_generator(data, self.\_cfg.learn.batch\_size, shuffle=True):  
 output = self.\_learn\_model.forward(batch['obs'], mode='compute\_actor\_critic')  
 adv = batch['adv'].squeeze(-1)  
 output['logit'] = output['logit'].squeeze(1)  
 batch['logit'] = batch['logit'].squeeze(1)  
 if self.\_adv\_norm:  
 # Normalize advantage in a train\_batch  
 adv = (adv - adv.mean()) / (adv.std() + 1e-5)

（5）将处理后的数据打包输入进行loss计算，clip\_ratio一般取0.2：

ppo\_batch = ppo\_data(output['logit'], batch['logit'], batch['action'], output['value'], batch['value'], adv,batch['return'], batch['weight'])  
 ppo\_loss, ppo\_info = ppo\_error(ppo\_batch, self.\_clip\_ratio)

ppo\_error 的具体步骤在本小节后的关键步骤中。

（6）第五步返回的loss有三种类型，分为policy\_loss，value\_loss和 entropy\_loss, 由于actor和critic网络使用了公共参数，并且可以通过添加熵加成来确保足够的探索性。根据PPO论文所指出的（如下图），loss计算公式应该为：  


wv, we = self.\_value\_weight, self.\_entropy\_weight  
 total\_loss = ppo\_loss.policy\_loss + wv \* ppo\_loss.value\_loss - we \* ppo\_loss.entropy\_loss  
 self.\_optimizer.zero\_grad()  
 total\_loss.backward()  
 self.\_optimizer.step()

其中，wv和we分别为value\_loss和entropy\_loss的权重，一般取0.5和0.01

### 关键步骤

#### 求adv(使用GAE方法)

（1）数据整理，代码如下：

compute\_adv\_data = gae\_data(value, next\_value, data['reward'], data['done'], data['traj\_flag'])  
 data['adv'] = gae(compute\_adv\_data, self.\_gamma, self.\_gae\_lambda) # 0.9, 0.95

（2）gae函数求每帧数据对应的adv：

def gae(data: namedtuple, gamma: float = 0.99, lambda\_: float = 0.97):  
 value, next\_value, reward, done, traj\_flag = data  
 if done is None:  
 done = torch.zeros\_like(reward, device=reward.device)  
 if len(value.shape) == len(reward.shape) + 1: # for some marl case: value(T, B, A), reward(T, B)  
 reward = reward.unsqueeze(-1)  
 done = done.unsqueeze(-1)  
 delta = reward + (1 - done) \* gamma \* next\_value - value  
 factor = gamma \* lambda\_  
 adv = torch.zeros\_like(value, device=value.device)  
 gae\_item = torch.zeros\_like(value[0])  
  
 for t in reversed(range(reward.shape[0])):  
 if traj\_flag is None: # trag\_flag is not None  
 gae\_item = delta[t] + factor \* gae\_item \* (1 - done[t])  
 else:  
 gae\_item = delta[t] + factor \* gae\_item \* (1 - traj\_flag[t].float())  
 adv[t] += gae\_item  
 return adv

关键的计算公式：

delta = reward + (1 - done) \* gamma \* next\_value - value

用reward、value和next\_value信息求的一个大致的基优势

gae\_item = delta[t] + factor \* gae\_item \* (1 - traj\_flag[t].float())

通过反向迭代，对基优势进行修正，修正后优势包含未来的奖励信息，其中

factor = gamma \* lambda\_

**factor越大，智能体越有远见，但相应模型越难收敛**

#### 求loss（policy\_loss, value\_loss, entropy\_loss)

##### （1）求三个loss，其中ppo\_output中包括ppo\_loss和entropy\_loss，ppo\_info中包括approx\_kl（新旧策略差异）和 clipfrac（clip的数据占比）。

def ppo\_error(  
 data: namedtuple,  
 clip\_ratio: float = 0.2,  
 use\_value\_clip: bool = True,  
 dual\_clip: Optional[float] = None  
) -> Tuple[namedtuple, namedtuple]:  
  
 assert dual\_clip is None or dual\_clip > 1.0, "dual\_clip value must be greater than 1.0, but get value: {}".format(dual\_clip)  
 logit\_new, logit\_old, action, value\_new, value\_old, adv, return\_, weight = data  
 policy\_data = ppo\_policy\_data(logit\_new, logit\_old, action, adv, weight)  
 policy\_output, policy\_info = ppo\_policy\_error(policy\_data, clip\_ratio, dual\_clip)  
 value\_data = ppo\_value\_data(value\_new, value\_old, return\_, weight)  
 value\_loss = ppo\_value\_error(value\_data, clip\_ratio, use\_value\_clip)  
  
 return ppo\_loss(policy\_output.policy\_loss, value\_loss, policy\_output.entropy\_loss), policy\_info

##### （2）ppo\_policy\_error()代码：

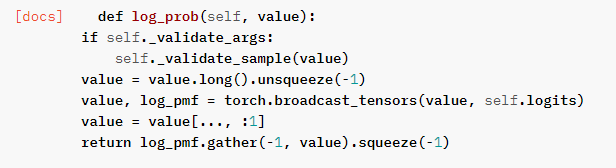
先给出ppo的loss公式：  
page6  
其中:  
page_7

代码：

def ppo\_policy\_error(data: namedtuple,  
 clip\_ratio: float = 0.2,  
 dual\_clip: Optional[float] = None) -> Tuple[namedtuple, namedtuple]:  
 logit\_new, logit\_old, action, adv, weight = data  
 if weight is None:  
 weight = torch.ones\_like(adv)  
 dist\_new = torch.distributions.categorical.Categorical(logits=logit\_new)  
 dist\_old = torch.distributions.categorical.Categorical(logits=logit\_old)  
 logp\_new = dist\_new.log\_prob(action)  
 logp\_old = dist\_old.log\_prob(action)  
 dist\_new\_entropy = dist\_new.entropy()  
 if dist\_new\_entropy.shape != weight.shape:  
 dist\_new\_entropy = dist\_new.entropy().mean(dim=1)  
 entropy\_loss = (dist\_new\_entropy \* weight).mean()  
 # policy\_loss  
 ratio = torch.exp(logp\_new - logp\_old)  
 if ratio.shape != adv.shape:  
 ratio = ratio.mean(dim=1)  
 surr1 = ratio \* adv  
 surr2 = ratio.clamp(1 - clip\_ratio, 1 + clip\_ratio) \* adv  
 if dual\_clip is not None: #不执行这里  
 clip1 = torch.min(surr1, surr2)  
 clip2 = torch.max(clip1, dual\_clip \* adv)  
 # only use dual\_clip when adv < 0  
 policy\_loss = -(torch.where(adv < 0, clip2, clip1) \* weight).mean()  
 else:  
 policy\_loss = (-torch.min(surr1, surr2) \* weight).mean()  
 with torch.no\_grad():  
 approx\_kl = (logp\_old - logp\_new).mean().item()  
 clipped = ratio.gt(1 + clip\_ratio) | ratio.lt(1 - clip\_ratio)  
 clipfrac = torch.as\_tensor(clipped).float().mean().item()  
 return ppo\_policy\_loss(policy\_loss, entropy\_loss), ppo\_info(approx\_kl, clipfrac)

其中关键代码：

dist\_new = torch.distributions.categorical.Categorical(logits=logit\_new)  
 dist\_old = torch.distributions.categorical.Categorical(logits=logit\_old)  
 logp\_new = dist\_new.log\_prob(action)  
 logp\_old = dist\_old.log\_prob(action)

torch.distributions.categorical.Categorical(logits=< >) 创建一个由logits参数组成的分布（logits是未标准化的概率），pytorch官方手册中是这样描述的  
  
dist\_new.log\_prob() 是用新旧策略对采样的动作进行处理**（这里具体处理理解的不是很清楚，其实就是对应了ppo算法中求新旧策略的差异）**并得到最后的logp\_new和logp\_old，（使用log\_prob是因为后边方便用对数运算的性质得到**新旧策略的比值作为新旧策略的差异**）  
log\_prob 源码：  
  
ratio就是PPO公式中的r(θ)，求法如下：

ratio = torch.exp(logp\_new - logp\_old)

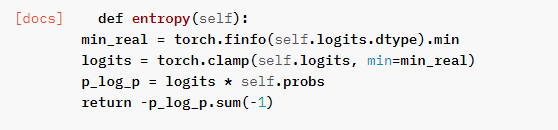
之后将r(θ)带入clip公式求ppo\_loss。weight通常是全为1向量。

policy\_loss = (-torch.min(surr1, surr2) \* weight).mean()

**至于ppo\_loss前为何加负号，看公式，由于使用了adv优势，所以我们希望得到更好的奖励，所以希望优势越高越好（最大化 torch.min(surr1, surr2)），根据神经网络的更新原理，可以reduce（-torch.min(surr1, surr2)）。**

此外，entropy\_loss的求解公式为:

dist\_new = torch.distributions.categorical.Categorical(logits=logit\_new)  
dist\_new\_entropy = dist\_new.entropy()  
entropy\_loss = (dist\_new\_entropy \* weight).mean()

先上pytorch官方源码：  


公式很这样看理解起来比较复杂，但是大体上是在求新策略的信息熵（entropy），**信息熵的定义是: 表示随机变量不确定的度量，越随机的信源熵越大。**也就是logit\_new（未标准化的概率，这段直接叫它概率）概率数组对应的动作不确定度。  
**例如，如果logit\_new 中每个下标对应的概率都一样，动作非常的不确定，那么它的entropy应该就很大，因为此时每个动作的概率都相等，所以logit\_new中相当于信息含量很少，它的entropy就很大；相反，在模型训练后期，智能体已经可以根据状态得到一个非常确定的动作了，也就是logit\_new中可能会有一个很大的概率和其他很小的概率，那么此时logit\_new中信息含量很大，它的entropy也就越小。**  
因此，希望entropy越来越小，所以loss的符号为+。（这部分是个人的一点理解，不知道是不是对的）

##### （3）ppo\_value\_error()代码：

def ppo\_value\_error(  
 data: namedtuple,  
 clip\_ratio: float = 0.2,  
 use\_value\_clip: bool = True,  
 ) -> torch.Tensor:  
 value\_new, value\_old, return\_, weight = data  
 if weight is None:  
 weight = torch.ones\_like(value\_old)  
 # value\_loss  
 if use\_value\_clip:  
 value\_clip = value\_old + (value\_new - value\_old).clamp(-clip\_ratio, clip\_ratio)  
 v1 = (return\_ - value\_new).pow(2)  
 v2 = (return\_ - value\_clip).pow(2)  
 value\_loss = 0.5 \* (torch.max(v1, v2) \* weight).mean()  
 else:  
 value\_loss = 0.5 \* ((return\_ - value\_new).pow(2) \* weight).mean()  
 return value\_loss

利用return和value\_new构成误差来求value\_loss，这里也用到了clip的思想，是为了防止网络更新过快，这里构建方法类似于DQN，就不多余赘述，value\_loss的符号为+。

## eval

（1）对data信息进行处理后（信息结构为：obs）输入网络求**action**和**logit**，代码如下：

with torch.no\_grad():  
 output = self.\_eval\_model.forward(data, mode='compute\_actor')

（2）与环境完成一个episode交互后求得评估信息并输出，输出内容如下（可参考使用）

'train\_iter': train\_iter, # 学习率  
 'ckpt\_name': 'iteration\_{}.pth.tar'.format(train\_iter),  
 'episode\_count': n\_episode,  
 'envstep\_count': envstep\_count,  
 'avg\_envstep\_per\_episode': envstep\_count / n\_episode,  
 'evaluate\_time': duration,  
 'avg\_envstep\_per\_sec': envstep\_count / duration,  
 'avg\_time\_per\_episode': n\_episode / duration,  
 'reward\_mean': np.mean(episode\_reward),  
 'reward\_std': np.std(episode\_reward),  
 'reward\_max': np.max(episode\_reward),  
 'reward\_min': np.min(episode\_reward),

# model:

## 主模型

主模型如下，其中**self.encoder**为对输入信息进行的特征提取，为Actor和Critic网络的共用部分，**self.actor\_head**为actor网络独有的输出部分，由于本环境使用的动作空间维度为16，因此输出的**logit（未归一化的概率）**也是16维的；**self.critic\_head** 为critic网络独有的部分，输出为评价值，维度为1。

mode = ['compute\_actor', 'compute\_critic', 'compute\_actor\_critic'] 分为三种网络输出模式，可以根据需要选择（官方库的设置非常灵活，可以记录一下这种forward选择方式）

class GoBiggerPPoModel(nn.Module):  
  
 mode = ['compute\_actor', 'compute\_critic', 'compute\_actor\_critic']  
  
 def \_\_init\_\_(self,  
 scalar\_shape: int,  
 food\_shape: int,  
 food\_relation\_shape: int,  
 thorn\_relation\_shape: int,  
 clone\_shape: int,  
 clone\_relation\_shape: int,  
 hidden\_shape: int,  
 encode\_shape: int,  
 action\_type\_shape: int,  
 rnn: bool = False,  
 critic\_head\_hidden\_size: int = 32,  
 critic\_head\_layer\_num: int = 1,  
 activation=nn.ReLU(inplace=True),  
 ) -> None:  
 super(GoBiggerPPoModel, self).\_\_init\_\_()  
 self.activation = activation  
 self.action\_type\_shape = action\_type\_shape  
 self.encoder = Encoder(scalar\_shape = scalar\_shape,  
 food\_shape = food\_shape,  
 food\_relation\_shape = food\_relation\_shape,  
 thorn\_relation\_shape = thorn\_relation\_shape,  
 clone\_shape = clone\_shape,  
 clone\_relation\_shape = clone\_relation\_shape,  
 hidden\_shape = hidden\_shape,  
 encode\_shape = encode\_shape,  
 activation = activation)  
  
 self.actor\_head = DiscreteHead(32, action\_type\_shape, layer\_num=2, activation=self.activation)  
 self.critic\_head = RegressionHead(critic\_head\_hidden\_size, 1, critic\_head\_layer\_num,activation=activation)  
  
 self.actor = [self.encoder, self.actor\_head]  
 self.critic = [self.encoder, self.critic\_head]  
  
 self.actor = nn.ModuleList(self.actor)  
 self.critic = nn.ModuleList(self.critic)  
  
 def forward(self, inputs, mode:str):  
  
 assert mode in self.mode, "not support forward mode:\ {}/{}".format(mode, self.mode)  
 return getattr(self, mode)(inputs)

## compute\_actor, compute\_critic, compute\_actor\_critic部分的代码：

### compute\_actor

def compute\_actor(self, inputs: torch.Tensor):  
 B = inputs['batch']  
 A = inputs['player\_num\_per\_team']  
  
 scalar = inputs['scalar']  
 food = inputs['food']  
 food\_relation = inputs['food\_relation']  
 thorn\_relation = inputs['thorn\_relation']  
 thorn\_mask = inputs['thorn\_mask']  
 clone = inputs['clone']  
 clone\_relation = inputs['clone\_relation']  
 clone\_mask = inputs['clone\_mask']  
  
 x = self.encoder(scalar, food, food\_relation, thorn\_relation, thorn\_mask, clone,clone\_relation, clone\_mask)  
 res = self.actor\_head(x)  
  
 action\_type\_logit = res['logit'] # B, M, action\_type\_size  
 action\_type\_logit = action\_type\_logit.reshape(B, A,\*action\_type\_logit.shape[1:])  
  
 return {'logit': action\_type\_logit,}

### compute\_critic：

def compute\_critic(self, inputs: torch.Tensor):  
 B = inputs['batch']  
 A = inputs['player\_num\_per\_team']  
 scalar = inputs['scalar']  
 food = inputs['food']  
 food\_relation = inputs['food\_relation']  
 thorn\_relation = inputs['thorn\_relation']  
 thorn\_mask = inputs['thorn\_mask']  
 clone = inputs['clone']  
 clone\_relation = inputs['clone\_relation']  
 clone\_mask = inputs['clone\_mask']  
  
 x = self.encoder(scalar, food, food\_relation, thorn\_relation, thorn\_mask, clone,clone\_relation, clone\_mask)  
 value = self.critic\_head(x)  
 value\_pred = value['pred']  
 value\_type\_pred = value\_pred.reshape(B, A, \*value\_pred.shape[1:])  
 value\_output\_pred = torch.mean(value\_type\_pred, 1).unsqueeze(-1)  
  
 return {'value': value\_output\_pred}

### compute\_actor\_critic:

def compute\_actor\_critic(self, inputs:torch.Tensor):
  
 B = inputs['batch']
  
 A = inputs['player\_num\_per\_team']
  
  
 scalar = inputs['scalar']
  
 food = inputs['food']
  
 food\_relation = inputs['food\_relation']
  
 thorn\_relation = inputs['thorn\_relation']
  
 thorn\_mask = inputs['thorn\_mask']
  
 clone = inputs['clone']
  
 clone\_relation = inputs['clone\_relation']
  
 clone\_mask = inputs['clone\_mask']
  
  
 actor\_embedding = critic\_embedding = self.encoder(scalar, food, food\_relation, thorn\_relation,thorn\_mask, clone, clone\_relation, clone\_mask)
  
  
 act = self.actor\_head(actor\_embedding)
  
 action\_logit = act['logit'] # B, M, action\_type\_size
  
 action\_type\_logit = action\_logit.reshape(B, A, \*action\_logit.shape[1:])
  
  
 value = self.critic\_head(critic\_embedding)
  
 value\_pred = value['pred']
  
 value\_type\_pred = value\_pred.reshape(B, A, \*value\_pred.shape[1:])
  
 value\_output\_pred = torch.mean(value\_type\_pred, 1).unsqueeze(-1)
  
  
 return {'logit': action\_type\_logit, 'value': value\_output\_pred}