PPO used in go-bigger

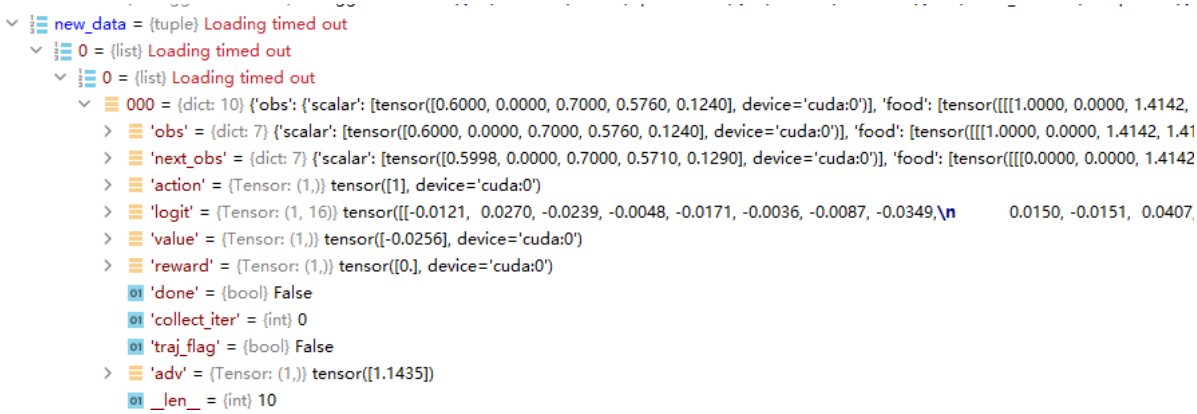
一般DRL在实际使用时根据流程可分类为collect、learn、eval三大过程，所以本文将PPO代码分成这三大过程逐一解析，重点解析learn部分的原理与代码，最后边再附上共享参数的PPO网络模型以供参考

collect

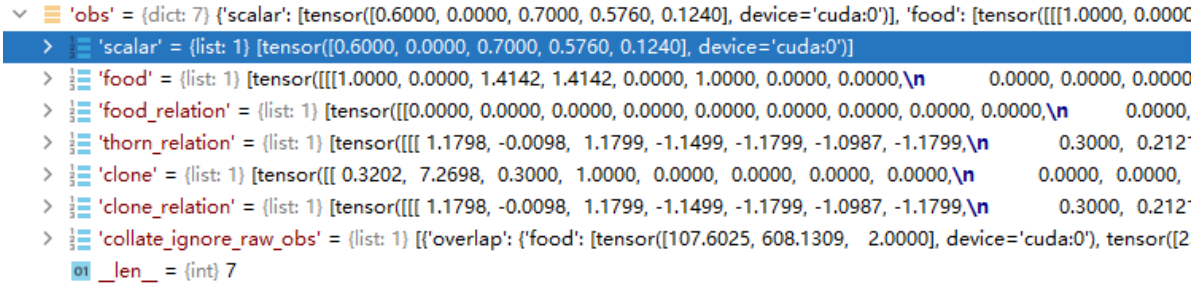
同时输出 action 和 value， 并且保存在buffer中：

```
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：

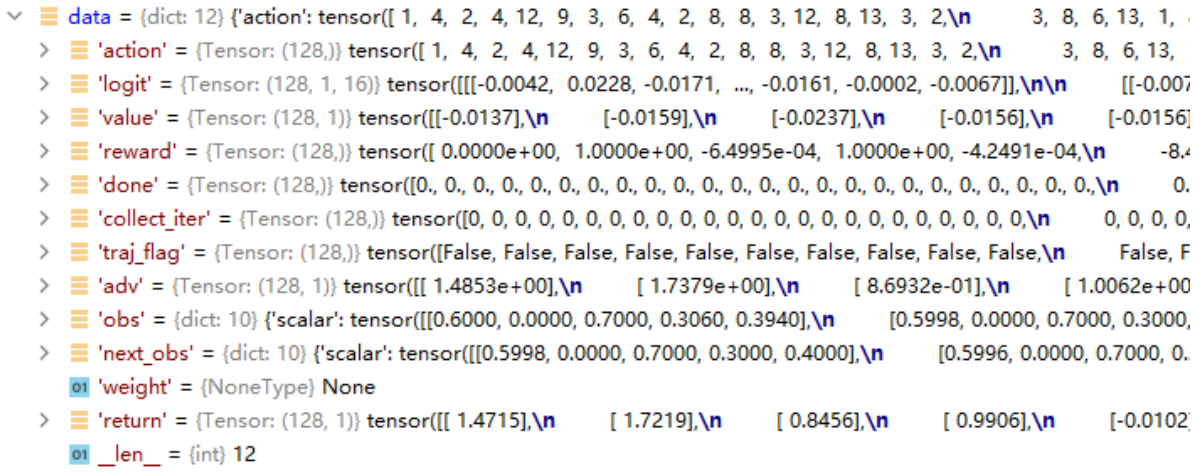
```
compute_adv_data = gae_data(value, next_value, data['reward'],
data['done'], data['traj_flag'])
data['adv'] = gae(compute_adv_data, self._gamma, self._gae_lambda)  # 0.9, 0.95
```

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：

finite-horizon estimators in [Mni+16]. If using a neural network architecture that shares parameters between the policy and value function, we must use a loss function that combines the policy surrogate and a value function error term. This objective can further be augmented by adding an entropy bonus to ensure sufficient exploration, as suggested in past work [Wil92; Mni+16]. Combining these terms, we obtain the following objective, which is (approximately) maximized each iteration:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right], \tag{9}$$

```
    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:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t\left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)\right]$$

其中:

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}$$

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

CLASS `torch.distributions.categorical.Categorical(probs=None, logits=None, validate_args=None)` [SOURCE]

Bases: `torch.distributions.distribution.Distribution`

Creates a categorical distribution parameterized by either `probs` or `logits` (but not both).

• NOTE

It is equivalent to the distribution that `torch.multinomial()` samples from.

Samples are integers from $\{0, \ldots, K-1\}$ where *K* is `probs.size(-1)`.

If *probs* is 1-dimensional with length-*K*, each element is the relative probability of sampling the class at that index.

If *probs* is N-dimensional, the first N-1 dimensions are treated as a batch of relative probability vectors.

• NOTE

The *probs* argument must be non-negative, finite and have a non-zero sum, and it will be normalized to sum to 1 along the last dimension. `probs` will return this normalized value. The *logits* argument will be interpreted as unnormalized log probabilities and can therefore be any real number. It will likewise be normalized so that the resulting probabilities sum to 1 along the last dimension. `logits` will return this normalized value.

See also: `torch.multinomial()`

Example:

```
>>> m = Categorical(torch.tensor([ 0.25, 0.25, 0.25, 0.25 ]))
>>> m.sample()  # equal probability of 0, 1, 2, 3
tensor(3)
```

Parameters

- **probs** (*Tensor*) – event probabilities
- **logits** (*Tensor*) – event log probabilities (unnormalized)

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：

```
[docs]    def log_prob(self, value):
        if self._validate_args:
            self._validate_sample(value)
        value = value.long().unsqueeze(-1)
        value, log_pmf = torch.broadcast_tensors(value, self.logits)
        value = value[..., :1]
        return log_pmf.gather(-1, value).squeeze(-1)
```

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：

```
[docs]    def entropy(self):
        min_real = torch.finfo(self.logits.dtype).min
        logits = torch.clamp(self.logits, min=min_real)
        p_log_p = logits * self.probs
        return -p_log_p.sum(-1)
```

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}
```