基于模糊逻辑规则集控制器初始化的 DDPG 控制器算法

Mars 董林森

2017.7.2

版本号 1.0

目录

1.算法模型：

算法模型实现了一种通用的控制器模型，该模型实现了两个功能，第一阶段为基于模糊逻辑规则初始化控制器，和第二阶段强化学习对控制器进行进一步优化。

该模型可以解决连续行为空间的控制问题，即控制的目标的行为空间存在连续型变量。

1.1 DDPG 模型

模型基于强化学习中的 DDPG（Deep Deterministic Policy Gradient）模型实现。在此基础上，通过使用模糊逻辑规则集对模型中的参数值进行初始化。

控制器的实现基于智能体（Agent）建模思想，将需要控制的实体抽象为 Agent，Agent 通过观测（Observation）和行为（Action）与外界环境交互，形成控制闭环。基于 Agent 的建模基本结构如下：

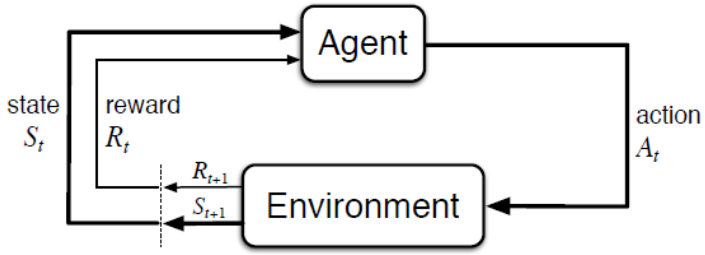

图 1

控制器的算法模型基于 DDPG 算法实现，主要包括两个神经网络，分别为 Actor 网络和 Critic 网络。其结构如下图：

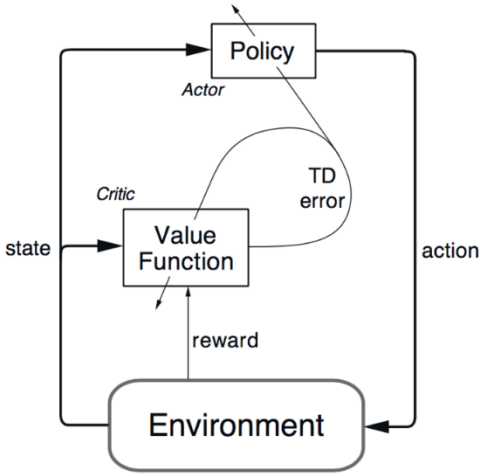

图 2

图 2 中，粗线条表示 DDPG 前向的控制阶段，细线条表示完成一次控制后，根据环境的奖赏进行学习的反向的学习阶段。

Actor 网络实现了一个强化学习中的策略函数（Policy Network），输入为某时刻的观测值，输出为在该观测值状态下的控制器的决策，即控制的行为，例如前进速度、方向等。Critic 网络实现了强化学习中状态-行为值函数（Q Network），即为对某个状态-行为组合好坏的评估。

1.2 模糊逻辑规则模型

同时在初始化阶段使用到基于模糊逻辑规则实现的控制器，表示了一种策略，即完成了从观测到行为的映射关系。

可以简单理解为如下形式：

其单条规则的基本规则的组成形式为

$$If\ \mathrm{f}(x_1\ \mathrm{is}\ A_1 \ldots x_k\ \mathrm{is}\ A_k)\ \ then\ \ y\ is\ B$$

y：称为后件，表示被推断出的值，即需要被决策的变量。

$x_1..x_1$：表示前件，表示实际观测到的某些变量的值，如温度，高度，速度等。

$A_1..A_k$ ：表示每个变量的模糊类别，或者称为 Linguistic label。每个变量x可包含多个类别，如：x 表示温度，则x对应的模糊类别的集合可以定义为 $A=\{高，中，低\}$，在任意一个x的值下，通过 Membership function 计算，可以得到变量x在所以类别下的真值，每一个真值都为一个[0 1]的连续值，应尽量保持在任意某个输入值下，对应的所有类别的真值的和为 1。通过 Membership function 的计算过程称为 Fuzzy 化的过程。

f ：连接所有前件变量的一个逻辑函数，它的输入为所有变量的在规则对应的模糊类别下的真值，输出为后件y为类别B的真值。

B：定义类似A，为输出变量的模糊类别。由f函数可以得到y在该类别下的真值，再通过对y的 Membership function 进行逆运算过程，求出y的实际值。逆运算的过程称为 Defuzzy，主要有 LMOM, TSK 等。

2.算法流程

这部分主要叙述基于模糊逻辑规则集初始化的 DDPG 控制器算法的流程。主要包括两步，第一通过模糊逻辑规则集对 DDPG 进行权值初始化的有监督学习过程；第二步为 DDPG 控制的强化学习过程。

2.1 输入输出的预处理

根据环境和被控制的实体，进行输入输出的抽象。若将整个模型看做一个控制器，则每个时刻，系统的输入为实际观测到的值，如敌方船只数量，我方船只数量。输出为需要被控制变量值，如速度的大小，前进的方向。

但不应该直接将模型设计为原始输入到原始输出的映射，如控制一个物体的移动时，可以发起两个控制指令，以某一速度前进或停止。如果直接将输出映射为两个变量，分别表示前进和停止，则并不是一个合理的设计，因为两个指令存在一定的逻辑关系，所以应抽象为对物体移动速度进行控制，当决策得到的速度为 0 时，即表示停止。

所以最终控制器的实际输入和输出都通过了一定的人为设计的预处理。类似如下结构

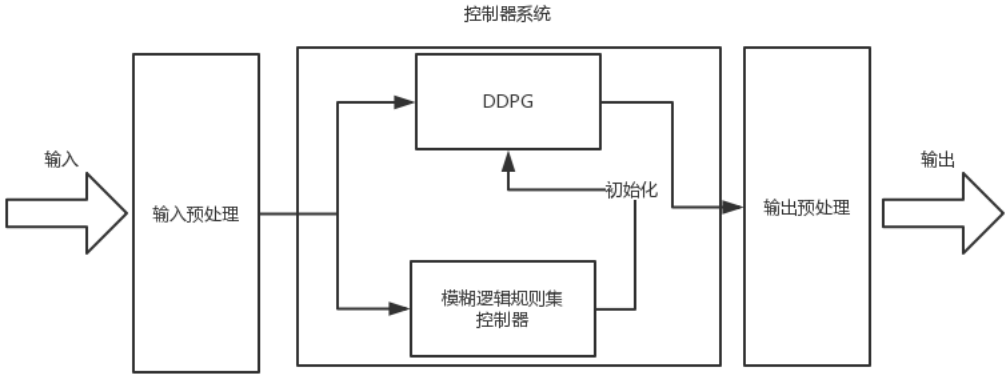

图 3

2.2 DDPG 的权值初始化

DDPG 模型包括两个神经网络 Actor 和 Critic，分别对其进行初始化，初始化的过程为可以看成有监督学习过程，过程不需要控制器和环境进行交互。

在确定模型的输入输出后，根据专家知识集等信息，分别产生两个模糊逻辑控制器，称为 Fuzzy Logic Controller 和 Fuzzy Logic Valuer。前者对应 Actor 网络的初始化，后者对应 Critic 网络的初始化。Fuzzy Logic Controller 对应 Actor 网络的初始化，其输入输出即对应图 3 控制器系统的输入输出，完成控制作用。Fuzzy Logic Valuer 类似强化学习中的 V 函数，对应 Critic 网络的初始化，输入某个时刻的状态，输出为该状态下最终累计奖赏值。因此由于算法的需

要，必须需要专家知识集提供能够生成这两种控制器的相关规则。

算法如下：

Algorithm: DDPG weight initialization

Define:

State Variable at time step is S_t

Action Variable at time step is A_t

Fuzzy logic controller is $FLC(S_t)$ and its output action is A_t^*

Fuzzy logic valuer is $FLV(S_t)$ andi its output value is V_t^*

Critic is $Q(S_t, A_t \mid \theta^Q)$, and its output is Q_t

Actor is $\mu(S_t \mid \theta^\mu)$, and its output is A_t

According to the DDPG algorithm, also create target network for critic and actor: $Q(S_t, A_t \mid \theta^{Q'})$ and $\mu(S_t \mid \theta^{\mu'})$

for epoch =1, M do:

Randomly generate N state S_t

Compute A_t^* and V_t^* according to $FLC(S_t)$ and $FLV(S_t)$

Compute A_t and according to $\mu(S_t)$

Compute Q_t in either $Q(S_t, A_t)$ or $Q(S_t, A_t^*)$

Update critic by minimizing the loss : $L = \frac{1}{N}\sum(V_t^* - Q_t)^2$

Update actor by minimizing the loss: $L = \frac{1}{N}\sum(A_t^* - A_t)^2$

Update target networks according the DDPG algorithm

$$\theta^{Q'} = \tau\theta^Q + (1-\tau)\theta^{Q'}$$

$$\theta^{\mu'} = \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

2.3 DDPG 强化学习

在完成初始化后，即可去掉模糊逻辑规则器部分，将 DDPG 放入环境中，进行强化学习。

DDPG 算法的学习算法如下图：

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

在进行强化学习时，由于涉及到多智能体协作问题，可以考虑将 DDPG 模型的强化学习模型做相应更改，该版本的实现中，暂时没有考虑因协作问题对算法做特殊化的涉及。

在强化学习过程中，需要注意环境的 Reward 的设计，它对模型的学习过程起到了决定性的作用。在设计 Reward 的计算方法时，应避免将关于策略的先验知识加入其中，即设计者不能通过 Reward 告诉控制器以怎样的方式去最大化累计奖赏。

2.4 控制器使用

在完成一定的训练后，即可将当前的网络模型，使用到实际场景中，此时不在进行学习行为。

3 代码文档

3.1 代码目录结构

GRAIC 主目录下主要包括 \data, \doc, \log, \src, \test 五个部分。

3.1.1 data

初始设计的思想是为了做基本的数据处理， 暂时未用到。所以该部分可以保留或删除。

3.1.2 doc

文档存放目录

3.1.3 log

日志存放目录，init.py 中获取了 LOG_PATH，为该目录的绝对路径，方便其他模块写入日志时获取地址。

初始化时产生的日志文件存在 log\initialTrain 目录下，根据每次运行时间创建一个目录，如 6-12-13-32-17 表示 6 月 12 日 13 点 32 分 17 秒。日志内容提供包括 loss.txt 存放了这次训练过程损失函数的值随训练次数的变化。Loss.txt 为一个 json 格式文件，为一个 list，每个元素为一个字典，结构如下：

```
{
    "Critic Cost": "10.20656",   // Critic 网络的损失函数值
    "epoch": 0,   //第几轮训练
    "Actor Cost": "10.24451"   // Actor 网路的损失函数值
},
```

log\initialTrain 还可以用于存储网络训练过程中的模型文件，调用 DDPGInitializer.save_all_model() 方法即可将训练时某一步的模型保存在对应的日志目录下。具体使用参考 DDPGInitializer 部分文档。

3.2 src

src__init__.py 中声明了 SRC_PATH，表示 src 目录的绝对路径。

3.2.1 Learner

Learner 部分实现了 DDPG 算法的神经网络和对对其初始化的方法。

3.2.1.1 Class Network(object)

表示网络模型的基类

Properties:

1. *sess*: 为该网络的 TensorFlow session

Methods:

1. *__init__(self, sess)*:
   使用一个 tensorflow session 初始化一个 Network，
2. variable(self, shape, f):
   返回一个形状为 shape 的 tf.Variable，并以根据 f 的值均匀随机初始化权值
3. batch_norm_layer(self, x, training_phase, scope_bn, activation=None):
   对 x 添加一个 batch norm 层，并根据是否在训练判断是否重用该层。

3.2.1.2 Class CriticNetwork(Network)

表示 DDPG 算法中 Critic 网络的类

Properties:

1. state_input:
   网络的 state 变量输入，类型为 tf.placeholder, 数据类型为 float
2. action_input:
   网络的 action 变量输入，类型为 tf.placeholder, 数据类型为 float
3. q_value_output:
   网络的 Q 值输出，为一个浮点数
4. target_state_input:
   属于 Target 网络，类比 state_input
5. target_action_input:
   属于 Target 网络，类比 action_input
6. target_q_value_output:
   属于 Target 网络，类比 q_value_output
7. target_update：
   对 target 网络的更新操作
8. target_is_training：
   表示 target 网络是否正在训练，参考 is_training
9. net:
   网络中所有的权值，为一个 python list
10. is_training:

表示是否正在训练的 bool 值，tf.placeholder，数据类型为 tf.bool

11. y_input:
    表示 DDPG 算法中，用于计算 loss 中的 y 的值，为一个 tf. placeholder
12. cost:
    表示 loss 函数的计算
13. optimizer:
    TensorFlow 中的优化器，优化算法为 AdamOptimizer
14. action_gradients:
    action 输出的梯度，用于对 actor 网络的更新，类型为 tf.gradients
15. time_step:
    用于记录训练次数的 int 值

Methods

1. __init__(self, sess, state_dim, action_dim):
   Critic 的构造函数，参数依次为 tf.Session，状态变量的形状，行为变量的形状
2. create_training_method(self):
   创建训练模型，主要包括 cost 和 optimizer，无返回值
3. create_q_network(self, state_dim, action_dim):
   创建 Q 网络，即 Critic 网络，返回值对应 Properties 中变量，依次为：state_input, action_input, q_value_output, net, is_training
4. create_target_q_network(self, state_dim, action_dim, net):
   创建 Target Q 网络，类比 network，不同在于 target 网络维护了对于 net 网络的权值的滑动平均值。具体参考 tf.train.ExponentialMovingAverage。返回值为 state_input, action_input, q_value_output, target_update, is_training。target_update 表示对 target 网络的更新操作。
5. update_target(self):
   更新 target 网络
6. train(self, y_batch, state_batch, action_batch):
   训练 Critic 网络，参数依次为 y，state，action。batch 大小必须保持相同。返回这一次训练的 cost。
7. gradients(self, state_batch, action_batch):
   返回 q 值对 action 的梯度值
8. target_q(self, state_batch, action_batch):
   计算并返回 target 网络对应 state 和 action 输入的 Q 值
9. q_value(self, state_batch, action_batch):
   计算并返回网络对应 state 和 action 输入的 Q 值

3.2.1.3 Class ActorNetwork(Network)

表示 DDPG 算法中 Actor 网络的类

Properties:

1. state_dim:
   状态变量的大小
2. action_dim

行为变量的大小

3. state_input:
   状态输入，类型为 tf.placeholder，由 self. create_network 返回
4. action_output:
   行为输入，类型为 tf.placeholder，由 self. create_network 返回
5. net:
   所有网络权值的 Python List 变量
6. is_training:
   表示是否正在训练的 bool 值，tf.placeholder，数据类型为 tf.bool
7. target_state_input:
   target 网络状态输入，类型为 tf.placeholder，由 self. create_target_network 返回
7. target_action_output:
   target 网络状态输入，类比 action_output
8. target_update:
   target 网络的更新操作
9. target_is_training:
   类比 is_training
10. q_gradient_input:
    通过 Critic 计算出的 Q 值函数对 action 的梯度值，为更新网络时需要的值。类型为 tf.placeholder
11. parameters_gradients:
    由 DDPG 中对 Actor 的更新公式得出网络参数的梯度值
12. optimizer:
    根据 parameters_gradients 值，生成的优化器。
13. y_input:
    用于网络初始化算法中的 y 值输入，作为 action 的标签信息。类型为 tf.placeholder，大小为 action_dim
14. cost:
    初始化算法中的 cost 计算
15. initial_optimizer:
    初始化算法中的优化器

Methods:

1. __init__(self, sess, state_dim, action_dim):
   构造函数
2. create_training_method(self):
   构造网络训练方法，无返回值
3. create_network(self, state_dim, action_dim):
   创建网络，依次返回 state_input, action_output, net, is_training，对应 property 中的变量
4. create_target_network(self, state_dim, action_dim, net):
   类似 create_network，依次返回 state_input, action_output, target_update, is_training，对应 target 中的变量。

5. update_target(self):
   更新 target 网络
6. train(self, q_gradient_batch, state_batch):
   训练 Actor 网络
7. initial_train(self, action_label_batch, state_batch):
   用于初始化算法中对 Actor 网络的训练。
8. actions(self, state_batch):
   用于训练过程中，输入 state_batch，返回 action_batch，is_training 设为 True
9. action(self, state):
   用于测试过程中，输入 state，返回 action，is_training 设为 False
10. target_actions(self, state_batch):
    返回 Target 网络的 action_batch，用于训练过程

3.2.1.4 Class NetworkCommon(object):

保存网络中的部分参数值，以及一些常用操作

Properties:

1. REPLAY_BUFFER_SIZE
2. REPLAY_START_SIZE
3. BATCH_SIZE
4. GAMMA Reward 的衰减系数

5. CRITIC_LAYER1_SIZE
6. CRITIC_LAYER2_SIZE
7. CRITIC_LEARNING_RATE
8. CRITIC_TAU 维持 Target 网络的滑动平均值的参数
9. CRITIC_L2 L2 正则化系数

10. ACTOR_LAYER1_SIZE
11. ACTOR_LAYER2_SIZE
12. ACTOR_LEARNING_RATE
13. ACTOR_TAU 维持 Target 网络的滑动平均值的参数
14. ACTOR_BATCH_SIZE
15. ACTOR_L2

Methods:

1. create_variable_summary(var, name_scope):
   用于保存变量的历史值，用于 tensorboard 的数据可视化使用
2. return_file_writer(sess, log_file_dir):
   返回一个目标文件为 log_file_dir 的 tf.summary.FileWriter，用于 tensorboard 的数据可视化使用

3.2.1.5 Class DDPGController(object):

表示 DDPG controller 的类，主要包括了 Critic 和 Actor 的实例，与环境的交互，人为添加的噪声用于环境的探索等操作

Properties:

1. action_dim:
   行为的大小
2. state_dim
   状态的大小
3. actor_network:
   一个 class ActorNetwork 的实例
4. critic_network:
   一个 class CriticNetwork 的实例
5. environment:
   一个 class Environment 的实例
6. exploration_noise:
   一个 class OUNoise 的实例，用于对 action 添加噪声，增加对环境的探索
7. model_saver:
   用于对 tensorflow 模型进行保存的 tf.train.Saver()
8. name:
   模型的名字，默认为 DDPG
9. replay_buffer:
   一个 class ReplayBuffer 的实例，参考 DDPG 算法中的 replay buffer 的作用
10. sess
    TensorFlow 的 session 变量

Methods:

1. __init__(self, env):
   传入一个 Environment 的实例，构造一个 DDPGController
2. train(self):
   训练控制器，先对 critic 和 actor 网络训练，再更新对应的 Target 网络
3. noise_action(self, state):
   返回添加噪声后的 actor 网络的 action 值，用于训练阶段获取 action
4. action(self, state):
   返回不添加噪声的 action 值，用于测试阶段
5. perceive(self, state, action, reward, next_state, done):
   DDPG 算法中，保存一次训练样本，并根据当前 replay_buffer 状态决定是否进行一次训练
6. initial_train(self, mini_batch):
   进行一次初始化算法中的训练。传入的 mini_batch 为一个 5*batch_size*?大小的数组，依次存储了 state_batch，action_batch，action_label_batch，value_label_batch，done_batch

7. save_model(self, path, check_point):
   保存 tf.session 中的网络模型保存在 path 路径中，参考 tf.train.Saver.save()方法。
8. load_model(self, path):
   载入 path 路径下的模型，参考 tf.train.Saver.restor()方法

Note：对于 DDPG 中输入的 state，必须先经过归一化到[0, 1]，然后输入到模型中，输出的 action 也为归一化值，所以需要转换为实际的值后再输入到环境中。

3.2.1.6 Class DDPGInitializer(object):

提供初始化算法中训练数据的产生，模型保存，模型初始化训练等

Properties:

1. ddpgController:
   需要被初始化的控制器，为 Class DDPGController 的实例
2. fuzzylogicController:
   用于初始化 DDPG 的模糊逻辑控制器 FLC
3. fuzzyLogicValuer:
   用于初始化 DDPG 的模糊逻辑 Valuer FLV
4. log_file_dir:
   用于保存日志文件的路径
5. model_file_dir:
   用于保存模型文件的路径
6. mini_batch_size:
   用于训练的数据的 batch_size 大小

Methods:

1. __init__(self, ddpg_controller,
   fuzzy_logic_controller,
   fuzzy_logic_valuer,
   mini_batch_size):
   构造函数，参数含义参考 property 部分
2. generate_state_done_mini_batch(self, mini_batch_size):
   按照均匀分布随机产生数量为 mini_batch_size 大小的 state 数据，数据已经归一化到[0, 1]
3. generate_training_sample_mini_batch(self, mini_batch_size):
   根据随机产生的 state 数据以及 FLC 和 FLV 产生训练数据，并返回
4. save_all_model(self, epoch):
   保存 DDPG 模型
5. load_model(self, path):
   载入 DDPG 模型
6. train_DDPG(self, epoch):
   基于初始化算法训练 DDPG 模型，并将训练过程的日志文件保存

3.2.1.7 Class OUNoise (object):

用于产生随机白噪声的类

Properties:

1. action_dimension:
2. mu:
3. sigma:
4. state:
5. theta:

Methods:

1. __init__(self, action_dimension, mu=0, theta=0.15, sigma=0.2):
2. reset(self):
   重置系统的状态
3. noise(self):
   返回随机白噪声的值

3.2.1.8 Class ReplayBuffer (object):

用于保存 DDPG 算法中的训练样本的 buffer

Properties:

1. buffer:
   一个 python deque，用于保存训练样本
2. buffer_size:
   buffer 大小
3. num_experiences:
   当前样本数目

Methods:

1. __init__(self, buffer_size):
2. get_batch(self, batch_size):
   返回一个 batch_size 大小的历史数据样本
3. size(self):
   返回 buffer_size
4. add(self, state, action, reward, new_state, done):
   添加一个新的样本，超过 buffer_size，则自动溢出队头的数据
5. count(self):
   返回当前历史数据样本数量
6. erase(self):
   清空 buffer

3.2.2 Fuzzy Logic Rule

3.2.2.1 Class Controller(object)

基于 fuzzy logic rule 实现的控制器，包括 FLC 和 FLV

Properties:

1. input_value_dict:
   存储控制器输入变量的字典，例如，输入变量有 'input_var_1', 'input_var_2',则该字典类似为：
   {
   'input_var_1': 0.5,
   'input_var_2':0.8
   }
2. output_value_dict:
   控制器输出变量的字典，类似 Input_value_dic
3. rule_section_set:
   存储 rule section 的 list
4. name:
   控制器的名称

Methods:

1. __init__(self, name):
2. add_rule_section(self, rule_set):
   添加一个 rule section
3. _assign_input_to_section(self):
   将输入变量字典中的值按照每个 rule section 各自的输入分别赋值，这么做的愿意在于每个 rule section 使用到的输入变量可能不同。
4. _calc_output(self):
   计算模型的输出

Note：input_var_dict 为一个 python property 类型，setter 中如果被设置了新的值，则自动计算其对应的输出。

3.2.2.2 Class Defuzzifier (object)

用于对输出变量去模糊化的基类

Properties:

1. name

Methods:

1. __init__(self, name):
2. defuzzify(self, degree, mf):
   参数为 degree 的值和一个 membership function，然后计算其真实值

Note：该类作为基类不应该被直接使用，应该使用其子类。

Note:省略 Defuzzifier 的子类的文档，包括：LMOMDefuzzifier, TSKDefuzzifier。具体计算方法的原理参见论文和资料

3.2.2.3 Class MembershipFunction (object)

成员函数类的基类

Properties:

1. name

Methods:

1. __init__(self, name):
2. calc(self, input_value):
   根据输入的值，计算其对应的逻辑值或者称为 degree
3. inverse_calc(self, input_degree):
   输入一个 degree，进行函数的逆运算，对于反函数存在多个值的情况，求对应的多值的平均值

Note: 省略 MembershipFunction 的子类的文档，包括 LeftTriangleMF, RightTriangleMF, TriangleMF, GaussianMF

3.2.2.4 Class Variable (object)

用于表示模糊逻辑中的变量

Properties:

1. name
2. mf
   成员函数的 python list，因为一个变量存在多个模糊类别所以为一个 list
3. value
   存储变量实际的值
4. degree
   计算变量在不同模糊类别下的逻辑值或称为 degree，为一个字典，key 为类别的名称，value 为对应的逻辑值
5. linguistic_label
   该变量的模糊类别的名称的 python list
6. upper_range
   变量 value 值的上界
7. lower_range
   变量 value 值的下界

Methods:

__init__(self, name, mf, range):

Note：value 和 degree 都被设置为 Python Property 类型

3.2.2.5 Class InputVariable (Variable)

表示输入变量

Properties:

Methods:

1. __init__(self, name, range, mf, value=0.0):
2. calc_degree(self):
   分别计算该输入变量在各个模糊类别的下的真值

Note：输入变量的 value 的 setter 被重载，当 value 被设置时，会自动调用 calc_degree()方法

3.2.2.6 Class OutputVariable (Variable)

表示输出变量

Properties:

1. defuzzifier
   一个 class Defuzzifier 的实例，用于对输出变量去模糊化

Methods:

1. _init__(self, name, mf, range, defuzzifier):

Note：degree 的 setter 属性被重载，当通过控制器计算出新的 degree 时，自动调用 self.defuzzifier.defuzzify()得到输出的真实值。

3.2.2.7 Class FuzzyRule(object)

表示一条模糊规则的类，包括存储这条规则的前件变量，后件变量

Properties:

1. rule_str:
   该规则的字符串表示，即”If……then……”。
2. true_value:
   这条规则在某个输入的下的真值
3. input_var_list:
   输入变量的 Python list，存储的为 inputVariable 类
4. input_dict:
   表示输入变量的值的字典，key 为变量的名称，value 为真实值
5. output_var_list:
   类比 input_var_list
6. output_var_value:
   输出变量的真实值
7. _min_operation:
   对前件逻辑值的最小化操作，有两种 soft_min 或 min
8. _section:
   该条规则所属的 section 的编号

Methods:

1. __init__(self,
   input_var_list,
   output_var_list,
   rule_str,
   section,
   min_operation="softmin"
   ):
2. set_input_var_value(self, new_input_var_value_dict):
   设置新的输入值，传入一个字典，key 为变量名，value 为值
3. set_input_output_dict(self, rule_str=None):
   分别设置新的输入输出的字典，即 input_dict 和 output_dict，传入参数为一个规则的 string
4. _get_true_value(self):
   计算该条规则的真值
5. set_output_var_degree(self):
   设置输出变量的 degree 为该条规则的真值，并通过去模糊化计算真实值后将其赋值给 self.output_var_value
6. _min_op(self):
   对输入真值的最小化操作，即取所有前件中最小的真值作为该条规则的真值
7. _softmin_op(self):
   对输入真值的软小化操作，即对 degree 使用负指数加权求均值。公式如下：

$$degree_{rule} = \frac{\sum degree_{var} * e^{-degree_{var}}}{\sum degree_{var}}$$

   其中$degree_{rule}$ 表示规则的真值， $degree_{var}$表示输入变量的真值

3.2.2.8 Class FuzzyRuleSet (object)

该类为多条规则的集合，即成为一个 Section

Properties:

1. input_var_value_dict:
   在该集合内的所有规则的前件的变量值的字典，key 为变量的名称，value 为真实值
2. name:
3. output_var_value_dict:
   类比 input_var_value_dict，规定一个 FuzzyRuleSet 中，所有的规则的输出有且仅有相同的一个，即一个 FuzzyRuleSet 控制系统的一个输出变量
4. rule_list:
   存储了该集合中所有规则的 Python list
5. rule_set_output_value:

表示对集合内所有规则计算后，得到的最终的该变量的输出值，计算方法为对集合内规则以该规则的真值为权值的加权平均值。

6. section:
   表示 section 的编号

Methods:

1. __init__(self, name, section, rule_list):
   构造函数，rule_list 为包含了 Class FuzzyRule 的列表
2. add_fuzzy_rule(self, fuzzy_rule):
   添加一条规则到实例中
3. get_input_var_name(self):
   将所有规则中的输入变量名称保存到一个列表中并返回
4. get_output_var_name(self):
   将所有规则中的输出变量名称保存到一个列表中并返回
5. assign_input_value(self):
   将 input_var_value_dict 中的值赋值给各条规则
6. _calc_value(self):
   计算最终输出量的值，并返回

3.2.2.9 Class RuleParser (object)

一个用于分解规则“if ……then……”字符串的类

Methods:

parse_if_then_rule(self, rule_str):

3.2.3 Environment

3.2.3.1 Class Environment(object)

描述环境的类

Properties:

1. action_dim:
   行为的大小
2. action_set:
   行为的集合，存储的类型为 Class Action
3. name:
4. reward:
   一个 Reward 的实例
5. state_dim:
   状态的大小
6. state_set:
   状态的集合，存储的类型为 Class State
7. time_step_limit:
   时间步数的限制

Methods:

1. __init__(self, name, state_set, action_set):
   构造一个环境实例
2. set_action_set(self, action_set):
   设置一个新的 action_set
3. set_reward(self, reward):
   设置一个新的 reward
4. set_state_set(self, state_set):
   设置一个新的 state_set
5. reset(self):
   重置环境
6. step(self, action):
   传入一个 action 类，执行环境模拟。返回值为 next_state, reward, done 依次表示下一个时刻状态，及时奖赏值，是否结束的标志
7. is_done(self, state):
   判断某个状态是否为一个合法的结束状态，返回一个 bool

Note: 关于 Environment 中 action，reward，state 的文档，由于代码比较简单，不再单独说明，直接参照源码即可。
Note: Environment 基类不应该直接被声明和使用，针对某个仿真环境或游戏环境，创建一个子类，并重载以上方法后使用。
Note:设计 Environment 类的想法在于，不管 DDPG 接入任何的仿真环境或模拟器中，其都应该提供以上方法，而这一过程通过实现一个 Environment 的子类进行包装和抽象。

3.3 test

test\ __init__.py 中声明了 TEST_PATH，表示 test 目录的绝对路径。

3.3.1 **initializationTest:**

initializationTest 用于测试使用模糊逻辑规则集对 DDPG 控制器神经网络权值进行初始化的方法。

3.3.1.1 class DDPGInitializaionTest(object)

用于测试的类，用于产生一些随机的数据和控制器等。不应该被其他模块使用和继承等。

Properties:

1. controller:
2. input_var_list:
3. output_var_list:
4. section_list:
5. section_num:

6. value:
7. value_rule:
8. valuer:

Methods:

1. __init__(self, state_dim, action_dim):
2. generate_input_var(self, input_dim=100):
   随机产生输入变量，返回的值为类型为 Class InputVariable 的列表
3. generate_output_var(self, output_dim=10):
   类比以上
4. generate_value(self):
   随机产生输入值
5. generate_section_list(self, rule_per_seciont_num=10):
   根据输入和输出变量构造 section 的列表
6. generate_fuzzy_logic_controller(self, output_dim=10):
   产生 FLC
7. generate_fuzzy_logic_valuer(self):
   产生 FLV

3.3.1.2 运行测试入口 main：

使用如下方式测试，参数 epoch 表示训练轮数，mini_batch 表示 mini_batch 大小

```
Usage:
    ddpginitializationTest.py epoch <epoch> mini_batch <mini_batch>

Options:
    -h --help
```

main 函数结构如下：

1. 依次声明
   a) 初始化测试实例 DDPGInitializaionTest
   b) 环境实例 Environment
   c) 两个模糊逻辑控制器实例 fuzzy_controller 和 fuzzy_valuer
   d) DDPG 初始化器的实例 ddpgInitializer

2. ddpgInitializer.train_DDPG(epoch=epoch) 对模型进行初始化训练
3. ddpgInitializer.save_all_model(epoch=epoch) 对模型进行保存