UDACITY

‹ Return to Classroom

DISCUSS ON STUDENT HUB

Teach a Quadcopter How to Fly

REVIEW

CODE REVIEW 1

HISTORY

▾ agents/agent.py 1

```
from agents.actor import Actor
from agents.critic import Critic
from agents.replay_buffer import ReplayBuffer
from agents.noise import Ornstein_Uhlenbeck_Process as noise
import numpy as np

class DDPG():
    def __init__(self, task):
        self.task = task
        self.state_size  = task.state_size
        self.action_size = task.action_size
        self.action_low  = task.action_low
        self.action_high = task.action_high

    def create_models(self, hidden_sizes_actor=(512,256), hidden_sizes_critic=(512,256,256)):
        self.actor_local  = Actor(self.state_size, self.action_size, self.action_low, self.action_high,
        self.actor_target = Actor(self.state_size, self.action_size, self.action_low, self.action_high,
        self.actor_target.model.set_weights(self.actor_local.model.get_weights())

        self.critic_local  = Critic(self.state_size, self.action_size, hidden_sizes=hidden_sizes_critic
        self.critic_target = Critic(self.state_size, self.action_size, hidden_sizes=hidden_sizes_critic
        self.critic_target.model.set_weights(self.critic_local.model.get_weights())

    def set_params(self,mu=0.1, sigma=0.1, theta=0.1, buffer_size=1e+8, batch_size=128, gamma=0.99, tau
        self.exploration_mu    = mu
        self.exploration_sigma = sigma
        self.exploration_theta = theta
        self.noise = noise(self.action_size, self.exploration_mu, self.exploration_theta, self.explorat
```

SUGGESTION

Tuning of noise parameters is quite important.. If your agent is exploring well, there is much chance to converge. Yo the noise is good for the action space or not. For example, if the action range is between (0-1), then good noise shou noise is distributed between (0.5 - 1), the quadcopter will never explore the action space below 0.5 using the noise.

```

        self.buffer_size = int(buffer_size)
        self.batch_size  = int(batch_size)
        self.buffer      = ReplayBuffer(self.buffer_size)

        self.gamma = gamma
        self.tau   = tau

    def act(self, states):
        state = np.reshape(states, [-1, self.state_size])
        action = self.actor_local.model.predict(state)[0]
        return list(action + self.noise.calc_noise())

    def learn(self):
        states, actions, rewards, dones, next_states = self.buffer.sample(self.batch_size, self.action_

        actions_next = self.actor_target.model.predict_on_batch(next_states)
        Q_targets_next = self.critic_target.model.predict_on_batch([next_states, actions_next])
        Q_targets = rewards + self.gamma * Q_targets_next * (1 - dones)

        self.critic_local.model.train_on_batch(x=[states, actions], y=Q_targets)

        action_gradients = np.reshape(self.critic_local.get_action_gradients([states, actions, 0]), (-1
        self.actor_local.train_fn([states, action_gradients, 1])

        # soft_update
        self.soft_update(self.critic_local.model, self.critic_target.model)
        self.soft_update(self.actor_local.model, self.actor_target.model)

    def reset_episode(self):
        self.noise.reset()
        state = self.task.reset()
        self.last_state = state
        return state

    def step(self, action, reward, next_state, done):
        self.buffer.add(self.last_state, action, reward, next_state, done)
        self.learn()
        self.last_state = next_state

    def soft_update(self, local_model, target_model):
        target_model.set_weights(self.tau * np.array(local_model.get_weights()) +
                                 (1 - self.tau) * np.array(target_model.get_weights()))
```

▸ utility_for_pycharm.py

▸ task.py

▸ physics_sim.py

▸ agents/replay_buffer.py

▸ agents/policy_search.py

▸ agents/noise.py

▸ agents/critic.py

▸ agents/actor.py