Adrian Chow

About

Solving Continuous Control using Deep Reinforcement Learning (Policy-Based Methods)

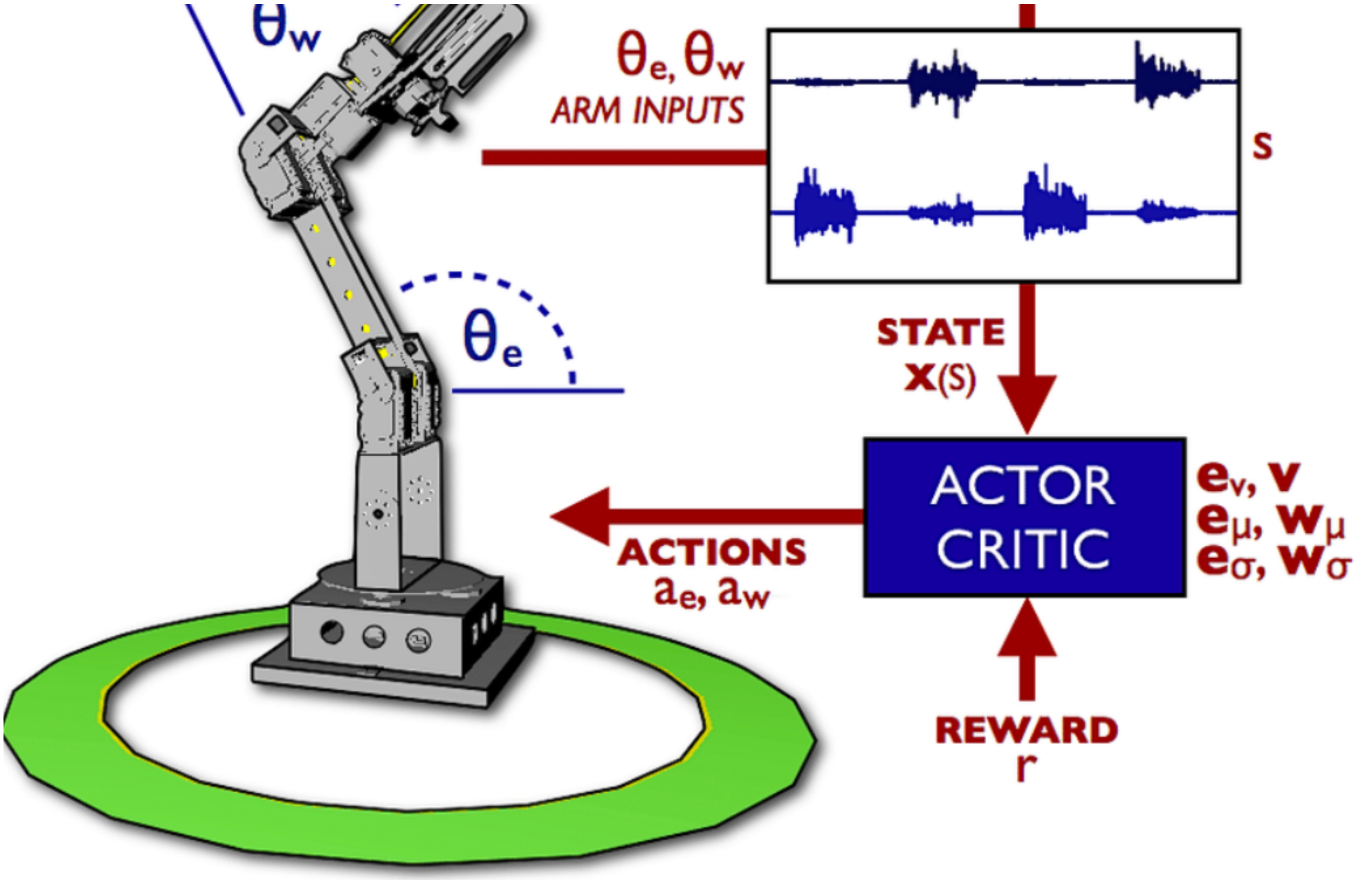
Adrian Chow Just now · 6 min read

Introduction

An alternative to classical control methods is deep reinforcement learning. Both are used to solve an optimization problem for dynamic systems that have a target behavior. **Classical control theory** deals with the behavior of dynamical systems with inputs and how behavior can be tuned by using feedback. On the other hand, **Deep RL**'s approach relies on an agent that is trained to have the policy which maximizes a measurable reward. The following article elaborates on a Deep RL agent's ability to solve a continuous control problem, namely Unity's Reacher.

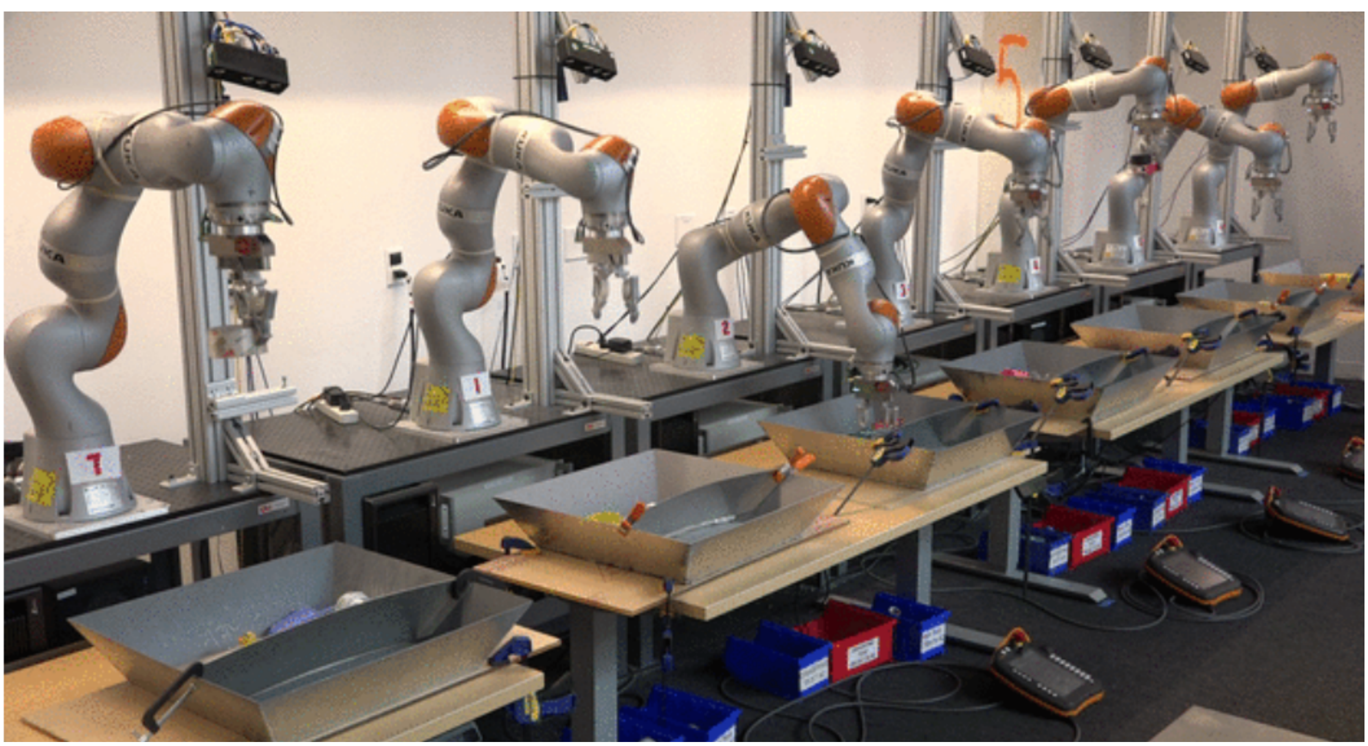
Real-World Robotics

The application of this project is targeted for, while not limited to, robotic arms. Once the control aspect of the stack is solved, one can then generate a behavior to plan tasks.



Robotic Arm using Deep Reinforcement Learning (Source: ResearchGate)

Furthermore, it is possible to increase the productivity of manufacturing by sharing the trained policy. It has been shown that having multiple copies of the same agent sharing experience can accelerate learning, as I learned from solving the project!

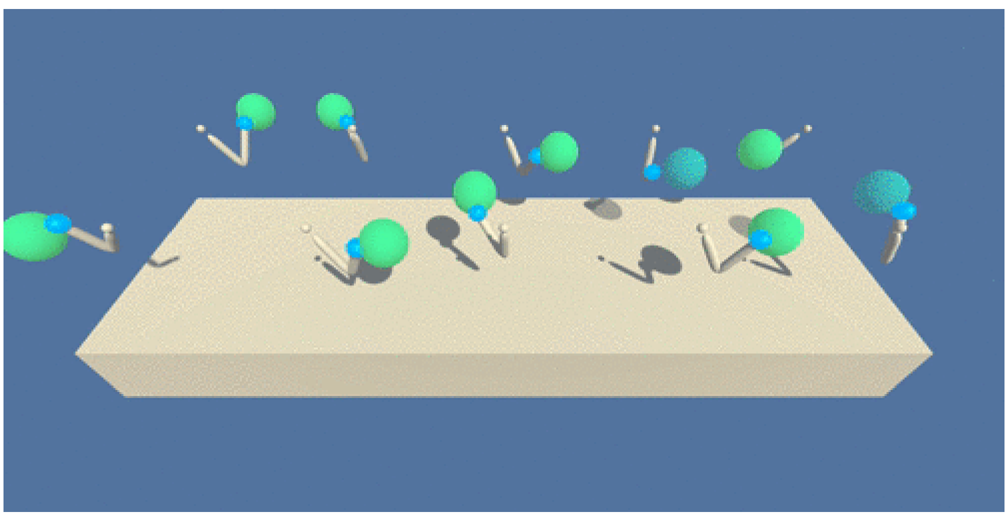


Multiple Robotic Arms using same Policy

The Environment

To solve the task of continuous control, I have used the Reacher environment on the Unity ML-Agents GitHub page.

In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of your agent is to maintain its position at the target location for as many time steps as possible.



Reacher Environment (Source: Unity ML-Agents)

RL Problem Specifications

- Goal of Agent: move to target locations
- Rewards : `+0.1` every step in the goal location, `+0.0` every step out of the goal location
- Action Space — Continuous, 4-Dimension Vector Space from [-1, +1]
- State Space — Continuous, 33-Dimension

- Solving Condition: Average score of +300.00 over 100 consecutive episodes.

Learning Algorithm

To solve the environment I have used an Actor-Critic Method which is a Deep Reinforcement Learning agent that utilizes two neural networks to estimate the policy (*actor*) and value function (*critic*). The policy structure is known as the *actor*, because it is used to select actions, and the estimated value function is known as the *critic* because it criticizes the actions made by the actor. Although I will go through the general algorithm, you can learn more about Actor-Critic Methods from Chris Yoon's in-depth article about Actor-Critic Methods. His work was tremendously useful when exploring this topic.

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

DDPG Algorithm (Source: Here)

The adaptation for the specific Actor-Critic method used is referred to as the Deep Deterministic Policy Gradient (DDPG). The original paper can be found here. The idea stems from the success of Deep-Q Learning and is modified to continuous action spaces.

The main takeaways are:

- Uses an Actor to choose the agent's actions deterministically.
- Uses a Critic to estimate the return value of the next state-action pair.
- DDPG is an Off-Policy Actor-Critic algorithm: Policy used to interact with the environment is different from the policy being learned.
- Uses soft-update to update target networks
- Uses some controlled action noise to explore action space

Initialization:

- Randomly initialize critic network Q(s, a|θ-Q) and actor μ(s|θ-μ) with weights θ-Q (critic) and θ-μ (actor).
- Initialize target networks denoted by Q' and μ' with weights θ-Q' ← θ-Q, θ-μ' ← θ-μ
- Initialize replay buffer R. The replay buffer is a data structure to This data structure that stores nodes (named tuples) with the following data: ["state", "action", "reward", "next_state", "done"]

Here is the implementation for the Actor and Critic Neural Networks.

Note that:

- The Actor uses the H**yperbolic Tangent Activation Function (tanh)** to limit the action space to [-1, 1].
- Using a seed will result in the same weights for initialization between the target and local model.
- Critic factors in the action during the second fully-connected layer.

```python
import numpy as np

import torch
import torch.nn as nn
import torch.nn.functional as F

def hidden_init(layer):
    fan_in = layer.weight.data.size()[0]
    lim = 1. / np.sqrt(fan_in)
    return (-lim, lim)

class Actor(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=128, fc2_units=128):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state):
        """Build an actor (policy) network that maps states -> actions."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return F.torch.tanh(self.fc3(x))


class Critic(nn.Module):
    """Critic (Value) Model."""
```

```

    def __init__(self, state_size, action_size, seed, fcs1_units=128, fc2_units=128):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fcs1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fcs1 = nn.Linear(state_size, fcs1_units)
        self.fc2 = nn.Linear(fcs1_units+action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fcs1.weight.data.uniform_(*hidden_init(self.fcs1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        """Build a critic (value) network that maps (state, action) pairs -> Q-values.""
        xs = F.relu(self.fcs1(state))
        x = torch.cat((xs, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

actor_critic_implementation.py hosted with ❤ by GitHub view raw

Here is the implementation of the Replay Buffer:

```
from collections import namedtuple, deque
import numpy as np
import random
import torch

field_names = ["state", "action", "reward", "next_state", "done"]

class ReplayBuffer:
    """ Fixed-size buffer to store experience tuples"""
```

```

    def __init__(self, action_size, buffer_size, batch_size, seed, device):
        """Initialize a ReplayBuffer object. """
        self.action_size = action_size
        self.buffer_size = buffer_size # size of replay buffer
        self.batch_size = batch_size # how many mem tuples to sample at a time
        self.seed = random.seed(seed)
        self.device = device

        # Define Named Tuple - field_names=["state", "action", "reward", "next_state", "
        self.experience = namedtuple("Experience", field_names=field_names)

        # Data structure to hold the memories
        self.memory = deque(maxlen=buffer_size)

    def add(self, state, action, reward, next_state, done):
        """Add a new experience to memory."""
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)


    def sample(self):
        """ Randomly sample a batch of experiences """

        # Sample an experience with length k from list of memories
        experiences = random.sample(self.memory, k=self.batch_size)

        # For each item in the tuple, stack vertically and convert to GPU torch tensor
        states = np.vstack([e.state for e in experiences if e is not None])
        states = torch.from_numpy(states).float().to(self.device) # (float)

        actions = np.vstack([e.action for e in experiences if e is not None])
        actions = torch.from_numpy(actions).float().to(self.device) # (float)

        rewards = np.vstack([e.reward for e in experiences if e is not None])
        rewards = torch.from_numpy(rewards).float().to(self.device) # (float)

        next_states = np.vstack([e.next_state for e in experiences if e is not None])
        next_states = torch.from_numpy(next_states).float().to(self.device) # float

        dones = np.vstack([e.done for e in experiences if e is not None]).astype(np.uint
        dones = torch.from_numpy(dones).float().to(self.device)

        return (states, actions, rewards, next_states, dones)

```

```

    def __len__(self):
        """Return the current size of internal memory."""
        return len(self.memory)
```

replay_buffer.py hosted with ❤ by GitHub view raw

Putting it together:

```
import numpy as np
import random
import copy
from collections import namedtuple, deque

from networks import Actor, Critic

import torch
import torch.nn.functional as F
import torch.optim as optim

BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR_ACTOR = 1e-3         # learning rate of the actor
LR_CRITIC = 1e-3        # learning rate of the critic
WEIGHT_DECAY = 0        # L2 weight decay

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class DDPG_Agent():
    """Interacts with and learns from the environment."""

    def __init__(self, state_size, action_size, n_agents, random_seed):
        """Initialize an Agent object.

        Params
        ======
            state_size (int): dimension of each state
            action_size (int): dimension of each action
            random_seed (int): random seed
        """
        self.state_size = state_size
        self.action_size = action_size
```

```
        self.action_size = action_size
        self.seed = random.seed(random_seed)

        # Actor Network (w/ Target Network)
        self.actor_local = Actor(state_size, action_size, random_seed).to(device)
        self.actor_target = Actor(state_size, action_size, random_seed).to(device)
        self.actor_optimizer = optim.Adam(self.actor_local.parameters(), lr=LR_ACTOR)

        # Critic Network (w/ Target Network)
        self.critic_local = Critic(state_size, action_size, random_seed).to(device)
        self.critic_target = Critic(state_size, action_size, random_seed).to(device)
        self.critic_optimizer = optim.Adam(self.critic_local.parameters(), lr=LR_CRITIC,

        # Noise process
        self.noise = OUNoise((n_agents, action_size), random_seed)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, random_seed)
```

DDPG_init.py hosted with ❤ by GitHub view raw

Training Loop

```
def follow_goal_ddpg(agent,
                    env,
                    brain_name,
                    n_agents,
                    n_episodes=1500,
                    max_t=3000,
                    print_every=10,
                    win_condition=30.0):

    """

    Continious Control using DDPG.

    Params
    ======
        n_episodes (int): maximum number of training episodes
        max_t (int): maximum number of timesteps per episode
```

```
        print_every (int): how many episodes before printing scores
        n_agents (int): how many arms are in the environment

    """

    scores = []
    scores_mean = []
    scores_window = deque(maxlen=100) # Score last 100 scores

    for i_episode in tqdm(range(1, n_episodes+1)):

        env_info = env.reset(train_mode=True)[brain_name] # reset the environment
        states = env_info.vector_observations             # get the current state (for e
        score = np.zeros(n_agents)                     # initialize the score (for each

        for t in range(max_t):
            actions = agent.act(states)                   # consult agent for actions
            env_info = env.step(actions)[brain_name]      # take a step in the env
            next_states = env_info.vector_observations    # get next state (for each age
            rewards = env_info.rewards                    # get reward (for each agent)
            dones = env_info.local_done                   # see if episode finished

            # take a learning step
            agent.step(states, actions, rewards, next_states, dones)

            score += env_info.rewards                    # update the score (for each ag
            states = next_states                          # roll over states to next tim
            if np.any(dones):                             # exit loop if episode finishe
                break

        scores.append(np.mean(score))
        scores_window.append(np.mean(score))
        scores_mean.append(np.mean(scores_window))

        # Print on print_every condition
        if i_episode % print_every == 0:
            print('\rEpisode {}\tScore: {:.2f}\tAverage Score: {:.2f}'.format(i_episode,

        # Winning condition + save model parameters
        if np.mean(scores_window) >= win_condition:
            print('\nEnvironment solved in {:d} episodes!\t Score: {:.2f} \tAverage Scor
            torch.save(agent.actor_local.state_dict(), 'checkpoint_actor.pth')
            torch.save(agent.critic_local.state_dict(), 'checkpoint_critic.pth')
            break

```

```
    execution_info = {'last_score': scores.pop(),
                      'solved_in': i_episode,
                      'last_100_avg': np.mean(scores_window),
                      'save_file': 'DDPG/...'}

    return scores, scores_mean, execution_info
```

Training_loop.py hosted with ❤ by GitHub view raw

For episode e ⟵1 to M:

- Initialize a random process N for action exploration
- Receive initial observation state, s1

```
for i_episode in range(1, n_episodes + 1):

   env_info = env.reset(train_mode=True)[brain_name]
   states = env_info.vector_observations
   score = np.zeros(n_agents)
```

Here we are able to sample our initial environment state as well as set up a list to store our training scores.

For step t ⟵ 1 to T:

```
for t in range(max_t):
    actions = agent.act(states)
    env_info = env.step(actions)[brain_name]
    next_states = env_info.vector_observations
    rewards = env_info.rewards
    dones = env_info.local_done
    agent.step(states, actions, rewards, next_states, dones)
    score += env_info.rewards
    states = next_states
    if np.any(dones): # Check if there are any done agents
        break
```

- Select action A= μ(st|θ-μ) + Noise according to the current policy and exploration noise

```
def act(self, state, add_noise=True):
    """Returns actions for given state as per current policy."""
    state = torch.from_numpy(state).float().to(device)
    self.actor_local.eval()
    with torch.no_grad():
        action = self.actor_local(state).cpu().data.numpy()
    self.actor_local.train()
    if add_noise:
        action += self.noise.sample()
    return np.clip(action, -1, 1)
```

get_action.py hosted with ❤ by GitHub view raw

Here we use the local Actor model to sample an action space with a bit of added noise exploration. For the noise added to the actions, DDPGs often use the Ornstein–Uhlenbeck process to generate temporally correlated exploration for exploration efficiency in physical control problems.

```
class OUNoise:
    """Ornstein-Uhlenbeck process."""

    def __init__(self, size, seed, mu=0., theta=0.15, sigma=0.2):
        """Initialize parameters and noise process."""
        self.size = size
        self.mu = mu * np.ones(size)
        self.theta = theta
        self.sigma = sigma
        self.seed = random.seed(seed)
        self.reset()

    def reset(self):
        """Reset the internal state (= noise) to mean (mu)."""
        self.state = copy.copy(self.mu)

    def sample(self):
        """Update internal state and return it as a noise sample."""
        x = self.state
        dx = self.theta * (self.mu - x) + self.sigma * np.random.standard_normal(self.si
        self.state = x + dx
        return self.state
```

Ornstein_Uhlenbeck_Process.py hosted with ❤ by GitHub view raw

- Execute action at and observe reward r(t) and observe new state s (t+1), where t is the current timestep

```
env_info = env.step(actions)[brain_name]
next_states = env_info.vector_observations
rewards = env_info.rewards
dones = env_info.local_done
```

- Store transition s(t), a(t), r(t), s(t+1)] in Replay Buffer, R

```
agent.step(states, actions, rewards, next_states, dones)
```

```
def step(self, states, actions, rewards, next_states, dones):
    """Save experience in replay memory, and use random sample from buffer to learn.""
    # Save experience / reward

    #for i in range(len(states)):
    #    self.memory.add(states[i, :], actions[i, :], rewards[i], next_states[i, :], c
    for state, action, reward, next_state, done in zip(states, actions, rewards, next_
        self.memory.add(state, action, reward, next_state, done)

    # Learn, if enough samples are available in memory
    if len(self.memory) > BATCH_SIZE:
        experiences = self.memory.sample()
        self.learn(experiences, GAMMA)
```

step.py hosted with ❤ by GitHub view raw

- The remaining parts are the algorithm are mathematical learning steps to update the policy (weights of the local and target networks).

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:

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

Learning Steps of DDPG Algorithm

The agent takes a step and immediately samples randomly from the replay buffer to learn a bit more about the environment it is currently in. We can split the update into two parts: Actor and Critic update. (Note: Actor loss is -ve due to the fact we are trying to maximize the return values passed by the Critic.)

DDPG only updates the local networks (the networks that interact with the environment) and performs soft updates on the target networks. As you can see below, soft updates only factor in a small amount of the local network weights (defined by Tau).

```
def learn(self, experiences, gamma):
    """Update policy and value parameters using given batch of experience tuples.
    Q_targets = r + γ * critic_target(next_state, actor_target(next_state))
    where:
        actor_target(state) -> action
        critic_target(state, action) -> Q-value
    Params
    ======
        experiences (Tuple[torch.Tensor]): tuple of (s, a, r, s', done) tuples
        gamma (float): discount factor
    """
    states, actions, rewards, next_states, dones = experiences

    # ---------------------------- update critic ---------------------------- #
    # Get predicted next-state actions and Q values from target models
    actions_next = self.actor_target(next_states)
    Q_targets_next = self.critic_target(next_states, actions_next)
    # Compute Q targets for current states (y_i)
    Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))
    # Compute critic loss
    Q_expected = self.critic_local(states, actions)
    critic_loss = F.mse_loss(Q_expected, Q_targets)
    # Minimize the loss
    self.critic_optimizer.zero_grad()
    critic_loss.backward()
    self.critic_optimizer.step()
```

```

    # ---------------------------- update actor ---------------------------- #
    # Compute actor loss
    actions_pred = self.actor_local(states)
    actor_loss = -self.critic_local(states, actions_pred).mean()
    # Minimize the loss
    self.actor_optimizer.zero_grad()
    actor_loss.backward()
    self.actor_optimizer.step()

    # ----------------------- update target networks ----------------------- #
    self.soft_update(self.critic_local, self.critic_target, TAU)
    self.soft_update(self.actor_local, self.actor_target, TAU)

def soft_update(self, local_model, target_model, tau):
    """Soft update model parameters.
    θ_target = τ*θ_local + (1 - τ)*θ_target
    Params
    ======
        local_model: PyTorch model (weights will be copied from)
        target_model: PyTorch model (weights will be copied to)
        tau (float): interpolation parameter
    """
    for target_param, local_param in zip(target_model.parameters(), local_model.paramete
        target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)
```

learn.py hosted with ❤ by GitHub view raw

Yeah, so that's it! The model is able to learn over time and develop the ability to follow a goal location in a continuously controlled state. The full repository can be viewed here.

Plot of Results

Training Parameters:

- Max Episodes: 1500
- Max Time Steps: 3000
- Buffer Size: 10000
- Batch Size: 128