DDPG Method

Presentation and Implementation

by Luca Marzari, University of Verona

20/05/2020

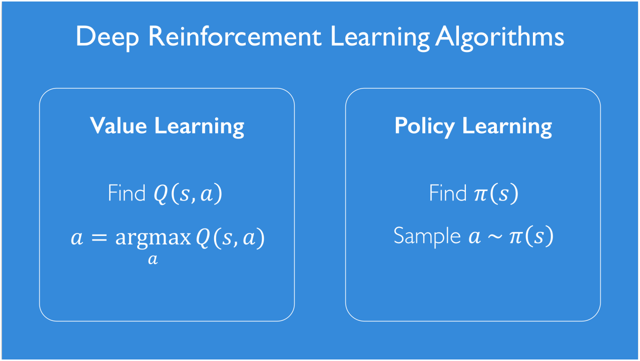
Content

- What is DDPG?
- Mathematical notations.
- From paper to code.
- Application: Fetch Pick and Place

What is DDPG?

where does it come from?

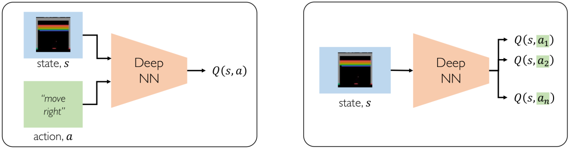
- Deep Deterministic Policy Gradients (DDPG) is algorithm introduced by Google Deepmind (Lillicrap et al, 2015) where they adapt the ideas underlying the success of Deep Q-Learning to the continuous action domain.
  They present an actor-critic, model-free algorithm based on the deterministic policy gradient that can operate over continuous action spaces.
- Actor-critic method is based on Values based method(DQN) and Policy gradient method.



- Q-learning uses value iteration to directly compute Q-values and find the optimal Q-function, but this approach becomes computationally inefficient and perhaps infeasible due to the computational resources and time this may take.
- Instead using this approach, is better use Deep Q-learning:

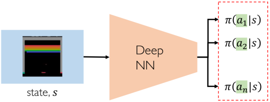
Deep Q Networks (DQN): Training

How can we use deep neural networks to model Q-functions?



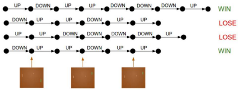
$$\mathcal{L} = \mathbb{E}\left[\left\| \overbrace{\left(r+\gamma\max_{a'}Q(s',a')\right)}^{\text{target}} - \overbrace{Q(s,a)}^{\text{predicted}} \right\|^2\right]$$

- DQN uses a function approximator to estimate the optimal Q-function, and to do that we use artificial neural networks. Receive as input a state S, and intrinsically compute for each pair (state, action) the Q-value.
- This approach of Deep Q-learning works efficiently for simple Video games(Atari), yet it struggles to find a convergent solution in continuous action space.



$\sum_{a_i \in A} \pi(a_i|s) = 1$

$\pi(a|s) = P(action|state)$



1. Run a policy for a while
2. Increase probability of actions that lead to high rewards
3. Decrease probability of actions that lead to low/no rewards

```
function REINFORCE
  Initialize θ
  for episode ~ π_θ
    {s_i, a_i, r_i}_{i=1}^{T-1} ← episode
    for t = 1 to T-1
      ∇ ← ∇_θ log π_θ(a_t|s_t) R_t
      θ ← θ + α∇
  return θ
```

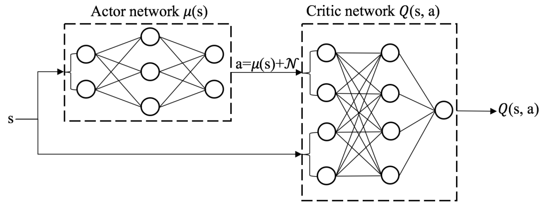
log-likelihood of action

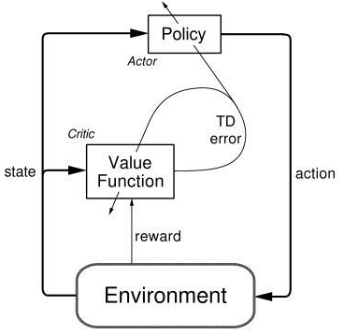
$$\nabla_\theta \log \pi_\theta(a_t|s_t)\, R_t$$

reward

- the goal is to directly learn a function that maps each state to an action. We directly optimise policy without using a value function. We use total rewards acquired in the episode as a measure of novelty of the policy
- problem: Since we must wait until the end of the episode to calculate the reward, we may conclude that, if we have a high reward R_{t,π_θ}, all the actions we took during the episode were good, even if some were bad.

- Actor Critic method is the combination of policy based methods, in particular of the REINFORCE algorithm, and value based methods





- The actor can be a function approximator like a neural network and its task is to produce the best action for a given state ($log_{\pi_\theta}(s, a)$)
- The critic is another function approximator, which receives as input the environment and the action by the actor, concatenates them and output the action value for the given pair ($Q_w(s, a)$).

First DDPG mathematical notation

Basic Elements:

- At each timestep t the agent receives an observation x_t, takes an action a_t and receives a scalar reward r_t
- The return from a state is defined as the sum of discounted future reward

$$R_t = \sum_{i=t}^{T} \gamma^{(i-t)} \ r(s_i, a_i), \quad \gamma \in [0, 1]$$

- action-value function describes the expected return after taking an action a_t in state s_t and thereafter following policy π ():

$$Q^{\pi}(s_t, a_t) = E[R_t | s_t, a_t]$$

Stochastic VS Deterministic policy

Stochastic policy is one in which the network maps the probability of taking an action to a given state. So input: set of states S, output: probability of selecting an action given a state.

Deterministic policy is one in which is one in which the network maps directly a state with an action.

How solve exploit/explore dilemma here? In this method they use a deterministic policy to avoid the inner expectation, and use another stochastic policy to avoid exploit/explore dilemma (off-policy learning)

If the target policy is deterministic we can describe it as a function $\mu : S \leftarrow A$:

$$Q^{\mu}(s_t, a_t) = E_{r_t, s_t+1}[r(s_t, a_t) + \gamma \ Q^{\mu}(s_{t+1}, \mu(s_{t+1})]$$

Loss function

- Q-learning (Watkins Dayan, 1992), a commonly used off-policy algorithm, uses the greedy policy $\mu(s) = arg\max_a Q(s, a)$
- They consider function approximators parameterized by θ^Q with this loss function:

$$
\begin{aligned}
L(\theta^Q) &= E\left[\left(Q(s_t, a_t|\theta^Q) - y_t\right)^2\right] \\
y_t &= r(s_t, a_t) + \gamma\, Q(s_{t+1}, \mu(s_{t+1})|\theta)
\end{aligned}
$$

- The use of large, non-linear function approximators for learning value or action-value functions has often been avoided in the past since theoretical performance guarantees are impossible, and practically learning tends to be unstable, so to solve this problem they use **replay buffer**, and a **separate target network for calculating** y_t

Algorithm

From paper to code

DDPG uses four neural networks:

- θ^Q Q network
- θ^μ deterministic policy network
- $\theta^{Q\prime}$ target Q network
- $\theta^{\mu\prime}$ a target policy network

The Q network and policy network is very much like simple Advantage Actor-Critic, but in DDPG, the Actor **directly** maps states to actions instead of outputting the probability distribution across a discrete action space.

The target networks are time-delayed copies of their original networks that slowly track the learned networks. Using these target value networks greatly improve stability in learning.
Here's why: In methods that do not use target networks, the update equations of the network are interdependent on the values calculated by the network itself, which makes it prone to divergence.

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 $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.autograd
from torch.autograd import Variable

class Actor(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, learning_rate = 3e-4):
        super(Actor, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.linear3 = nn.Linear(hidden_size, output_size)

    def forward(self, state):
        """
        Param state is a torch tensor
        """
        x = F.relu(self.linear1(state))
        x = F.relu(self.linear2(x))
        x = torch.tanh(self.linear3(x))

        return x

class Critic(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Critic, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.linear3 = nn.Linear(hidden_size, output_size)

    def forward(self, state, action):
        """
        Params state and actions are torch tensors
        """
        x = torch.cat([state, action], 1)
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = self.linear3(x)

        return x

```

Initialization networks:

```python
class Agent:
    def __init__(self, env, hidden_size=256, actor_learning_rate=1e-4, critic_learning_rate=1e-3, gamma=0.99, tau=1e-2, max_memory_size=50000):
        self.env = env
        # Params
        self.num_states = env.observation_space.shape[0] # number of states
        self.num_actions = env.action_space.shape[0] # number of actions
        self.gamma = gamma
        self.tau = tau

        # Networks
        self.actor = Actor(self.num_states, hidden_size, self.num_actions) # neural network of input num_states, hidden layer size hidden_size and output num_actions
        self.actor_target = Actor(self.num_states, hidden_size, self.num_actions) # same as above
        self.critic = Critic(self.num_states + self.num_actions, hidden_size, self.num_actions) # neural network of input num_states + num_actions, hidden layer size hidden_size and
                                                                                                 # output num_actions
        self.critic_target = Critic(self.num_states + self.num_actions, hidden_size, self.num_actions)

        for target_param, param in zip(self.actor_target.parameters(), self.actor.parameters()):
            target_param.data.copy_(param.data)

        for target_param, param in zip(self.critic_target.parameters(), self.critic.parameters()):
            target_param.data.copy_(param.data)
```

Learning process:

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

Replay Buffer

- Initialize replay buffer R: we use a replay buffer to solve the instability of function approximators for learning value. During each trajectory roll-out, we save all the experience tuples $(s_t, a_t, r_t, s_{t+1}, done)$ and store them in a finite-sized "cache".
- Then, we sample random mini-batches of experience from the replay buffer when we update the value and policy networks.
- we use experience replay because we want the data to be independently distributed. This fails to be the case when we optimize a sequential decision process in an on-policy way, because the data then would not be independent of each other. When we store them in a replay buffer and take random batches for training, we overcome this issue.

```python
class ReplayBuffer:
    def __init__(self, max_size):
        self.max_size = max_size
        self.buffer = deque(maxlen=max_size)

    def push(self, state, action, reward, next_state, done):
        experience = (state, action, np.array([reward]), next_state, done)
        self.buffer.append(experience)

    def sample(self, batch_size):
        state_batch = []
        action_batch = []
        reward_batch = []
        next_state_batch = []
        done_batch = []

        batch = random.sample(self.buffer, batch_size)

        for experience in batch:
            state, action, reward, next_state, done = experience
            state_batch.append(state)
            action_batch.append(action)
            reward_batch.append(reward)
            next_state_batch.append(next_state)
            done_batch.append(done)

        return state_batch, action_batch, reward_batch, next_state_batch, done_batch

    def __len__(self):
        return len(self.buffer)
```

Training part

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

```python
    def train(self, max_episode, max_step, batch_size, env):
        rewards = []

        for episode in range(max_episode):
            self.noise.reset()
            state = self.env.reset()
            episode_reward = 0

            for step in range(max_step):
                #env.render()
                action = self.get_action(state)
                action = self.noise.get_action(action, step)
                new_state, reward, done, _ = self.env.step(action)
                self.replay_buffer.push(state, action, reward, new_state, done)

                if len(self.replay_buffer) > batch_size:
                    self.update(batch_size)

                state = new_state
                episode_reward += reward

                if done:
                    print("episode " + str(episode) + ", " + "reward  " + str(episode_reward))
                    break

            rewards.append(episode_reward)

        return rewards
```

Update part

1 Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
2 Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
3 Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
4 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:

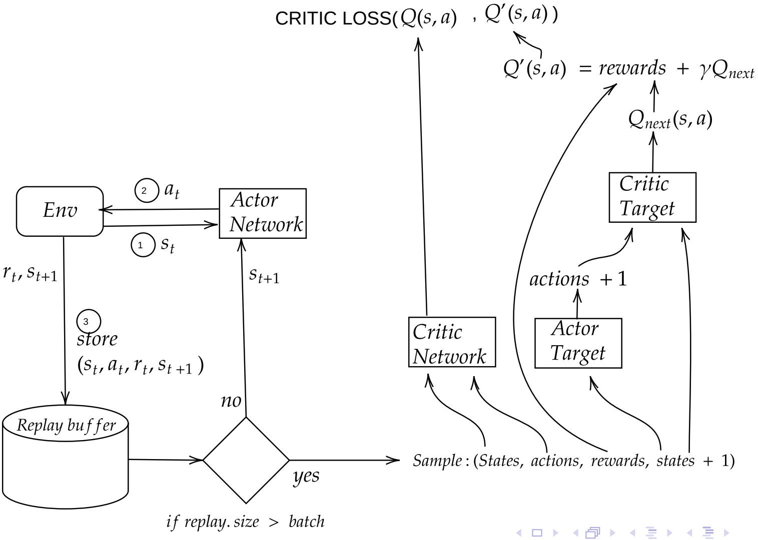
5

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

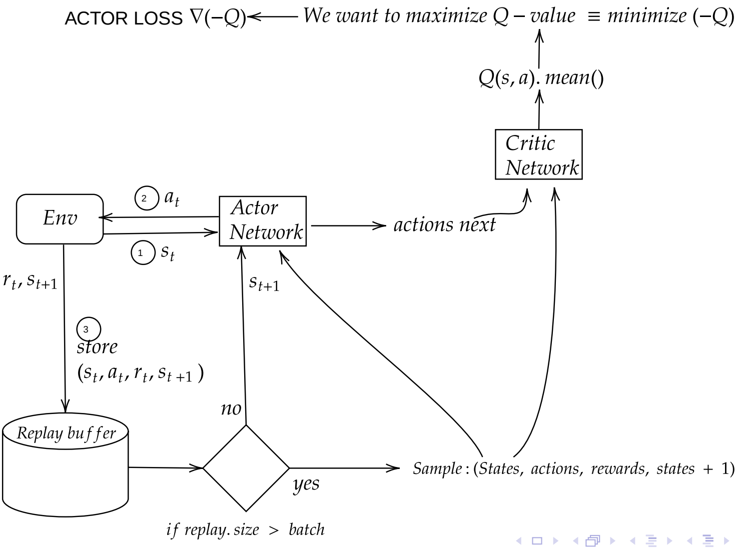
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

```python
    def update(self, batch_size):
        states, actions, rewards, next_states, _ = self.replay_buffer.sample(batch_size) # take sta
        states = torch.FloatTensor(states) # transform a list in a FloatTensor --> a tensor of floa
        actions = torch.FloatTensor(actions)
        rewards = torch.FloatTensor(rewards)
        next_states = torch.FloatTensor(next_states)

        # Critic loss --> check notes pag 2 we want to minimize the loss function
        Qvals = self.critic.forward(states, actions) # this is the q value output of critic network
        next_actions = self.actor_target.forward(next_states) # this are the actions from output of
        next_Q = self.critic_target.forward(next_states, next_actions.detach()) # combine actor(nex
        Qprime = rewards + self.gamma * next_Q # calculate Qprime

        critic_loss = self.critic_loss(Qvals, Qprime) # the difference between qvalue - q'

        # Actor loss
        policy_loss = -self.critic.forward(states, self.actor.forward(states)).mean()

        # update networks
        self.actor_optimizer.zero_grad() # gradient descent
        policy_loss.backward() # backpropagation
        self.actor_optimizer.step() # step

        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        # update target networks
        for target_param, param in zip(self.actor_target.parameters(), self.actor.parameters()):
            target_param.data.copy_(param.data * self.tau + target_param.data * (1.0 - self.tau))

        for target_param, param in zip(self.critic_target.parameters(), self.critic.parameters()):
            target_param.data.copy_(param.data * self.tau + target_param.data * (1.0 - self.tau))
```

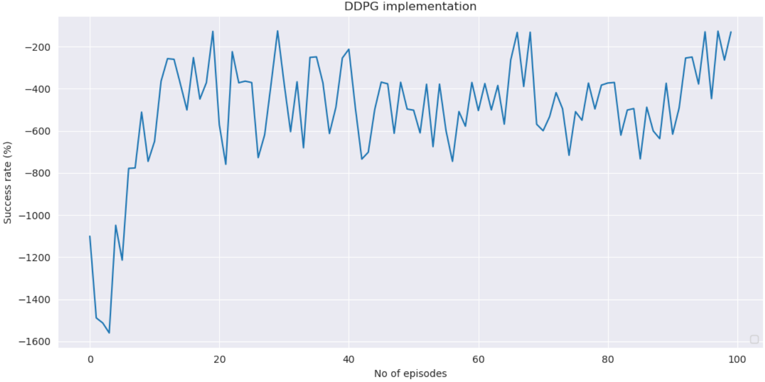
Flow chart Critic Loss



Flow chart Actor Loss



Plot DDPG implementation on "Pendulum-v0"



The End