Deep Deterministic Policy Gradient (DDPG)

Overview

Deep Deterministic Policy Gradient (DDPG) is an off-policy reinforcement learning algorithm designed for environments with continuous action spaces. It combines the deterministic policy gradient theorem with deep function approximators, enabling efficient learning in high-dimensional action spaces. (spinningup.openai.com)

Why Use DDPG?

DDPG is particularly effective in scenarios where actions are continuous, such as robotic control and autonomous driving. Its ability to handle high-dimensional action spaces makes it suitable for complex tasks where traditional discrete action methods are not applicable.

Prerequisites

To implement DDPG, ensure the following Python packages are installed:

- **PyTorch**: For building and training neural networks.
- **OpenAI Gym**: For creating and interacting with various environments.

Install them using pip:

```
pip install torch gym
```

Files Included

- **Python Script**: Contains the implementation of the DDPG algorithm, including the actor and critic networks, replay buffer, and training loop.

Code Description

1. **Import Libraries**:

   ```
    import torch
   import torch.nn as nn
   import torch.optim as optim
   import numpy as np
   import gym
   ```

   Imports necessary libraries for tensor operations, neural network construction, optimization, and environment interaction.

2. **Define Actor and Critic Networks**:

```
class Actor(nn.Module):
    def __init__(self, state_dim, action_dim, max_action):
        super(Actor, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(state_dim, 400),
            nn.ReLU(),
            nn.Linear(400, 300),
            nn.ReLU(),
            nn.Linear(300, action_dim),
            nn.Tanh()
        )
        self.max_action = max_action

    def forward(self, state):
        return self.max_action * self.model(state)
```

The **Actor** network outputs actions based on the current state, scaled by the maximum action value.

```
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(state_dim + action_dim, 400),
            nn.ReLU(),
            nn.Linear(400, 300),
            nn.ReLU(),
            nn.Linear(300, 1)
        )

    def forward(self, state, action):
        return self.model(torch.cat([state, action], 1))
```

The **Critic** network evaluates the action taken by the Actor by estimating the Q-value.

3. **Initialize Networks and Optimizers**:

```
 actor = Actor(state_dim=3, action_dim=1, max_action=1)
critic = Critic(state_dim=3, action_dim=1)
actor_target = Actor(state_dim=3, action_dim=1, max_action=1)
critic_target = Critic(state_dim=3, action_dim=1)
actor_optimizer = optim.Adam(actor.parameters(), lr=1e-3)
critic_optimizer = optim.Adam(critic.parameters(), lr=1e-3)
```

Creates instances of the Actor and Critic networks, along with their target networks and optimizers.

4. **Define Replay Buffer**:

```
class ReplayBuffer:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buffer = []
        self.position = 0

    def push(self, state, action, reward, next_state, done):
        if len(self.buffer) < self.capacity:
            self.buffer.append(None)
        self.buffer[self.position] = (state, action, reward, next_state, done)
```

```
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        batch = np.random.choice(self.buffer, batch_size, replace=False)
        state, action, reward, next_state, done = zip(*batch)
        return (
            torch.tensor(state, dtype=torch.float32),
            torch.tensor(action, dtype=torch.float32),
            torch.tensor(reward, dtype=torch.float32),
            torch.tensor(next_state, dtype=torch.float32),
            torch.tensor(done, dtype=torch.float32)
        )
```

The **ReplayBuffer** stores experiences and samples random batches for training.

5. **Training Loop**:

```
def train():
    replay_buffer = ReplayBuffer(1000000)
    state = env.reset()
    episode_reward = 0
    for t in range(1000000):
        action = actor(torch.tensor(state, dtype=torch.float32))
        next_state, reward, done, _ = env.step(action.detach().numpy())
        replay_buffer.push(state, action.detach().numpy(), reward, next_state, done
        state = next_state
        episode_reward += reward

        if len(replay_buffer.buffer) > 1000:
            batch = replay_buffer.sample(100)
            state_batch, action_batch, reward_batch, next_state_batch, done_batch =

            # Update Critic
            target_q = reward_batch + (1 - done_batch) * 0.99 * critic_target(next_
            current_q = critic(state_batch, action_batch)
            critic_loss = nn.MSELoss()(current_q, target_q)
            critic_optimizer.zero_grad()
            critic_loss.backward()
            critic_optimizer.step()

            # Update Actor
            actor_loss = -critic(state_batch, actor(state_batch)).mean()
            actor_optimizer.zero_grad()
            actor_loss.backward()
            actor_optimizer.step()

            # Update Target Networks
            for target_param, param in zip(actor_target.parameters(), actor.paramet
                target_param.data.copy_(0.995 * target_param.data + 0.005 * param.d
            for target_param, param in zip(critic_target.parameters(), critic.param
                target_param.data.copy_(0.995 * target_param.data + 0.005 * param.d

        if done:
            state = env.reset()
            episode_reward = 0
```

The **train** function interacts with the environment, stores experiences, and updates the networks using the DDPG algorithm.

Expected Output

The training loop does not produce immediate output but updates the Actor and Critic networks over time. Monitoring the cumulative reward per episode can provide insights into the agent's learning progress.

Use Cases

DDPG is suitable for tasks involving continuous action spaces,

---

References

Deep Deterministic Policy Gradient (DDPG) is an off-policy reinforcement learning algorithm designed for environments with continuous action spaces. It combines the deterministic policy gradient theorem with deep function approximators, enabling efficient learning in high-dimensional action spaces. (spinningup.openai.com)

References:

- Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Silver, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.
- OpenAI Spinning Up in Deep RL. (n.d.). Deep Deterministic Policy Gradient. Retrieved from (spinningup.openai.com)
- Yoon, C. (2019). Deep Deterministic Policy Gradients Explained. *Towards Data Science*. Retrieved from (towardsdatascience.com)
- MathWorks. (n.d.). Deep Deterministic Policy Gradient (DDPG) Agent. Retrieved from (mathworks.com)
- Wikipedia contributors. (2023). Model-free (reinforcement learning). *Wikipedia*. Retrieved from (en.wikipedia.org)
- Wikipedia contributors. (2023). Actor-critic algorithm. *Wikipedia*. Retrieved from (en.wikipedia.org)