Project Documentation - Code Analysis & Breakdown

Step 1: Install Dependencies

The first step is to install the required external libraries for running the project. This can be done through entering a command like below in one's terminal. PyBullet is essential for physics simulations in this implementation.

Code Reference:

```
pip install pybullet
```

Explanation:

- **PyBullet**:
  - Provides a physics engine for reinforcement learning tasks.
  - Enables simulation of environments like Ant, Half-Cheetah, and Humanoid.

Step 2: Import Libraries

The necessary libraries include tools for mathematical computations, neural network construction, and environment interaction.

Code Reference:

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import gym
import os
```

import time

Explanation:

1. `numpy`:
   - Handles array operations and mathematical computations.
2. `torch`:
   - Core library for building and training neural networks.
   - Supports GPU acceleration for faster computation.
3. `torch.nn` **and** `torch.nn.functional`:
   - Provides layers and activation functions for defining neural networks.
4. `gym`:
   - Framework for reinforcement learning environments.
5. `os`:
   - Manages file paths and directories for saving models and results.
6. `time`:
   - Measures the duration of training and evaluation.

Step 3: Actor and Critic Networks (Detailed Simplification)

These networks are central to the TD3 algorithm, where the Actor decides actions, and the Critic evaluates those actions.

Actor Network

The Actor network is like the decision-maker. It takes the current state of the environment (a description of everything happening) and decides the next move.

Code Reference

python

```
class Actor(nn.Module):
      def __init__(self, state_dim, action_dim, max_action):
            super(Actor, self).__init__()
            self.layer_1 = nn.Linear(state_dim, 400)
            self.layer_2 = nn.Linear(400, 300)
            self.layer_3 = nn.Linear(300, action_dim)
            self.max_action = max_action
      def forward(self, x):
            # Pass through first layer and apply ReLU
            x = F.relu(self.layer_1(x))
            # Pass through second layer and apply ReLU
            x = F.relu(self.layer_2(x))
            x = self.max_action * torch.tanh(self.layer_3(x))
        return x
```

Breaking It Down

1. State Input
    - The state (state_dim) is a set of numbers describing the environment.
    - Example: For a robot walking, the state might include angles of joints or speed.
2. Network Layers
    - Layer 1: First step to process the state. It has 400 neurons (units that learn patterns).
    - Layer 2: Builds on Layer 1, refining the understanding further with 300 neurons.
    - Layer 3: Produces the action (action_dim), deciding what the agent should do.
    - These layers are like steps in thinking: analyze, refine, decide.
3. Activation Functions
    - ReLU (Rectified Linear Unit)

        - Ensures non-negative outputs, making learning more stable.
        - Example: If the raw output of a layer is -5, ReLU converts it to 0.
    - Tanh (Hyperbolic Tangent)
        - Produces outputs between -1 and 1.
        - Useful for actions that have limits (e.g., how far a robot can turn).
4. Output Scaling
    - The final action is scaled by max_action, ensuring it stays within allowed limits.
    - Example: If max_action = 2, the output action will always be between -2 and 2.

Critic Network

The Critic network judges the Actor's decisions by estimating the quality of a state-action pair. It answers: "How good is this action for this state?"

Code Reference

python

```
class Critic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()
        self.layer_1 = nn.Linear(state_dim + action_dim, 400)
        self.layer_2 = nn.Linear(400, 300)
        self.layer_3 = nn.Linear(300, 1)
        # Second Critic network
        self.layer_4 = nn.Linear(state_dim + action_dim, 400)
        self.layer_5 = nn.Linear(400, 300)
        self.layer_6 = nn.Linear(300, 1)
    def forward(self, x, u):
        # Combine state and action
        xu = torch.cat([x, u], 1)
        # First Critic network
        x1 = F.relu(self.layer_1(xu))
```

```
x1 = F.relu(self.layer_2(x1))
x1 = self.layer_3(x1)
# Second Critic network
x2 = F.relu(self.layer_4(xu))
x2 = F.relu(self.layer_5(x2))
x2 = self.layer_6(x2)
return x1, x2
```

Breaking It Down

1. Inputs
   - State (x): The same description of the environment used by the Actor.
   - Action (u): The action chosen by the Actor.
   - These are combined (torch.cat) so the Critic can judge them together.
2. Two Critic Networks
   - There are two separate neural networks (Critic 1 and Critic 2) with the same structure
     - Input: Combined state and action.
     - Layers: 400 → 300 → 1.
     - Output: A single Q-value (score) representing how good the action is.
   - Having two Critics helps avoid overestimating Q-values, which can lead to bad decisions.
3. Outputs
   - forward: Returns Q-values from both Critics.
   - Example: If Critic 1 says the Q-value is 5, and Critic 2 says it's 6, both are returned.
   - Q1: Returns only the output of Critic 1, used for updating the Actor.

Key Concepts

1. Actor and Critic Work Together
    - The Actor tries to maximize the Critic’s Q-values by picking better actions.
2. Two Critics Reduce Errors
    - By comparing two Critics, the algorithm avoids being too optimistic about bad actions.

Step 4: Replay Buffer

The Replay Buffer is like the agent's memory. It stores past experiences so the agent can learn from them later. Without this, the agent would forget what it learned, and training would be unstable.

Replay Buffer

The Replay Buffer stores information about the environment, including the state, action, reward, and the next state. This allows the agent to:

- Revisit old experiences for learning.
- Learn from a variety of examples, not just the most recent ones.

Code Reference:

python

```
class ReplayBuffer:
      def __init__(self):
            self.storage = []
      def add(self, data):
            self.storage.append(data)
      def sample(self, batch_size):
            ind = np.random.randint(0, len(self.storage), size=batch_size)
            batch_states, batch_next_states, batch_actions, batch_rewards,
            batch_dones = [], [], [], [], []
            for i in ind:
                  state, next_state, action, reward, done = self.storage[i]
                  batch_states.append(np.array(state, copy=False))
                  batch_next_states.append(np.array(next_state, copy=False))
                  batch_actions.append(np.array(action, copy=False))
                  batch_rewards.append(np.array(reward, copy=False))
                  batch_dones.append(np.array(done, copy=False))
            return (
                  np.array(batch_states),
                  np.array(batch_next_states),
                  np.array(batch_actions),
                  np.array(batch_rewards).reshape(-1, 1),
                  np.array(batch_dones).reshape(-1, 1),
            )
```

Breaking It Down:

1. Initialization:
    - self.storage: A list to store experiences (state, action, reward, next state, done).
2. Adding Experiences:
    - add(data):
        - Adds a tuple (state, next_state, action, reward, done) to the memory.

Example:

python

```
replay_buffer.add((state, next_state, action, reward, done))
```

Sampling Experiences:

- `sample(batch_size)`:
    - Randomly selects a batch of experiences to train on.
    - Why random? So the agent doesn't learn patterns specific to the order of experiences.
    - Steps:
        1. Pick random indices (ind).
        2. Collect states, actions, rewards, etc., for those indices.
        3. Return the batch as arrays.
- **Batch Output:**
    - Returns five arrays:
        - `batch_states`: States sampled from memory.
        - `batch_next_states`: The states resulting from the actions.
        - `batch_actions`: Actions taken.
        - `batch_rewards`: Rewards received for those actions.
        - `batch_dones`: Flags (True/False) for whether the episode ended.

Why It's Important:

- Breaks Correlation: By randomly sampling, the agent avoids learning biased sequences.
- Reuses Data: The buffer allows the agent to reuse experiences for better learning efficiency.
- Stabilizes Training: Instead of relying only on new data, the agent learns from a mix of past and recent experiences.

Example Workflow:

1. Agent Adds Experience:

After each action, the agent saves the result:

python

```
replay_buffer.add((state, next_state, action, reward, done))
```

2. Agent Samples a Batch:

Before training, it grabs random experiences:

python

```
states, next_states, actions, rewards, dones =
replay_buffer.sample(batch_size)
```

3. Training Happens:

- The batch is used to update the Actor and Critic networks.

Key Concepts:

1. Memory: Replay Buffer is a memory bank for the agent.
2. Random Sampling: Makes learning less biased and more robust.
3. Batch Training: Allows the agent to learn efficiently by processing multiple examples at once.

Step 5: TD3 Algorithm Overview

The Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm improves on the standard Deep Deterministic Policy Gradient (DDPG) algorithm by addressing its key weaknesses. Here’s how TD3 works and what makes it special.

What Does TD3 Do?

TD3 is used for continuous action spaces, meaning environments where actions aren't just "left" or "right" but can take any value (e.g., controlling a robot arm). It improves stability and performance by:

1. Using Two Critic Networks: Reduces overestimation of Q-values.
2. Adding Target Policy Smoothing: Adds small noise to actions to prevent overfitting.
3. Delaying Updates to the Actor: Updates the Actor network less often than the Critic networks for better stability.

Key Steps in TD3

Let's break the algorithm into steps, showing how it's implemented in the code.

---

1. Initialize Networks

TD3 uses two Critics and one Actor, along with their target networks (slightly slower copies used for stable learning).

Code Reference:

python

```
class TD3(object):
      def __init__(self, state_dim, action_dim, max_action):
            self.actor = Actor(state_dim, action_dim, max_action).to(device)
            self.actor_target = Actor(state_dim, action_dim,
            max_action).to(device)
            self.actor_target.load_state_dict(self.actor.state_dict())
            self.critic = Critic(state_dim, action_dim).to(device)
            self.critic_target = Critic(state_dim, action_dim).to(device)
            self.critic_target.load_state_dict(self.critic.state_dict())
            self.actor_optimizer = torch.optim.Adam(self.actor.parameters())
            self.critic_optimizer = torch.optim.Adam(self.critic.parameters())
```

```
        self.max_action = max_action
```

Breaking It Down:

1. Actor and Critics:
    - The Actor predicts actions based on the current state.
    - Two Critics evaluate the quality of actions using Q-values.
2. Target Networks:
    - actor_target and critic_target are slightly delayed versions of the main networks.
    - These targets are updated slowly to stabilize training.
3. Optimizers:
    - Adam optimizer adjusts network weights during training.

---

2. Select Actions

The Actor outputs an action for a given state. TD3 adds noise to this action for exploration.

Code Reference:

python

```
def select_action(self, state):
    state = torch.Tensor(state.reshape(1, -1)).to(device)
    return self.actor(state).cpu().data.numpy().flatten()
```

Breaking It Down:

1. Input: The current state.
2. Output: An action predicted by the Actor.

Exploration: In training, Gaussian noise is added to the action:

python

```
action = action + np.random.normal(0, noise_std, size=action.shape)
```

---

3. Train the Networks

This is where the magic happens. The agent learns by using data from the Replay Buffer.

Code Reference:

python

```
def train(self, replay_buffer, iterations, batch_size, discount, tau,
policy_noise, noise_clip, policy_freq):
      for it in range(iterations):
            # Sample batch of transitions
            state, next_state, action, reward, done =
            replay_buffer.sample(batch_size)

            # Compute target Q-value
            next_action = self.actor_target(next_state)
            noise = torch.normal(0, policy_noise,
            size=next_action.shape).clamp(-noise_clip, noise_clip).to(device)
            next_action = (next_action + noise).clamp(-self.max_action,
            self.max_action)
            target_Q1, target_Q2 = self.critic_target(next_state, next_action)
            target_Q = reward + (1 - done) * discount * torch.min(target_Q1,
            target_Q2).detach()

            # Update Critic
            current_Q1, current_Q2 = self.critic(state, action)
            critic_loss = F.mse_loss(current_Q1, target_Q) +
            F.mse_loss(current_Q2, target_Q)
            self.critic_optimizer.zero_grad()
            critic_loss.backward()
            self.critic_optimizer.step()
```

```
# Delayed Actor and Target Updates
if it % policy_freq == 0:
actor_loss = -self.critic.Q1(state, self.actor(state)).mean()
self.actor_optimizer.zero_grad()
actor_loss.backward()
self.actor_optimizer.step()

# Update target networks
for param, target_param in zip(self.critic.parameters(),
self.critic_target.parameters()):
      target_param.data.copy_(tau * param.data + (1 - tau) *
      target_param.data)
for param, target_param in zip(self.actor.parameters(),
self.actor_target.parameters()):
    target_param.data.copy_(tau * param.data + (1 - tau) *
      target_param.data)
```

Breaking It Down:

1. Sample Batch:
   - A batch of experiences is sampled from the Replay Buffer.
2. Target Q-Value:
   - Predict the Q-value for the next state and action.
   - Take the minimum of the two Critics' predictions (torch.min) to reduce overestimation.
3. Update Critic:
   - Compare the target Q-value to the current Q-value.
   - Update the Critic networks using the Mean Squared Error (MSE) loss.
4. Delayed Actor Update:
   - The Actor is updated less frequently (e.g., every two iterations).
   - It uses the Critic's Q-values to improve its actions.

5. Update Target Networks:
    - Slowly update the target networks using Polyak Averaging:

python

```
target_param.data.copy_(tau * param.data + (1 - tau) * target_param.data)
```

Key TD3 Innovations

1. Twin Critics:
    - Two Critics help reduce overestimation of Q-values.
2. Policy Smoothing:
    - Adds noise to the target policy, making learning more robust.
3. Delayed Updates:
    - Actor updates are less frequent, stabilizing training.

Why TD3 is Better

- Improves Stability: Reduces overestimation errors.
- Enhances Learning: Smooth policies and delayed updates prevent the agent from making erratic decisions.
- Efficient Training: Uses Replay Buffer and batch processing to train on diverse data.

Step 6: Integration with the Environment

In reinforcement learning, the agent interacts with an environment to learn. In TD3, we use environments like OpenAI Gym or PyBullet. These simulate real-world tasks, such as controlling a robot or balancing a pole, allowing the agent to practice.

How Does Integration Work?

The agent:

1. Observes the state: A snapshot of the environment.
2. Chooses an action: Based on its Actor network.
3. Receives feedback: The environment gives a reward and the next state.
4. Learns from experience: Stores the experience in the Replay Buffer and uses it for training.

Setting Up the Environment

The code begins by creating and configuring the environment.

Code Reference:

python

```
env_name = "AntBulletEnv-v0"  # Name of the environment
env = gym.make(env_name)  # Initialize the environment

state_dim = env.observation_space.shape[0]  # Size of the state
action_dim = env.action_space.shape[0]  # Size of the action
max_action = float(env.action_space.high[0])  # Maximum action value
```

Breaking It Down:

1. Environment Name:
   - AntBulletEnv-v0 simulates an ant-like robot that learns to walk.
   - This can be replaced with other environments like "HalfCheetah-v2" or "Humanoid-v2".
2. Observation Space:
   - state_dim: Number of features describing the environment.
   - Example: For the Ant, the state might include joint angles and velocities.
3. Action Space:
   - action_dim: Number of possible actions.

- max_action: Maximum value for any action.
- Example: For the Ant, actions control the force applied to each joint.

Main Training Loop

The agent interacts with the environment over multiple steps, learning from its experiences.

Code Reference:

python

```
total_timesteps = 0
done = True
while total_timesteps < max_timesteps:
      if done:
            # Reset environment when an episode ends
            obs = env.reset()
            done = False
            episode_reward = 0
            episode_timesteps = 0

      # Select action
      if total_timesteps < start_timesteps:
            # Random action for exploration
            action = env.action_space.sample()
      else:
            # Use Actor network
            action = policy.select_action(np.array(obs))
            action = (action + np.random.normal(0, expl_noise,
            size=env.action_space.shape[0])).clip(env.action_space.low,
            env.action_space.high)

      # Perform action in the environment
```

```
new_obs, reward, done, _ = env.step(action)

# Store transition in Replay Buffer
replay_buffer.add((obs, new_obs, action, reward, float(done)))

# Update state and counters
obs = new_obs
total_timesteps += 1
episode_timesteps += 1
episode_reward += reward
```

Breaking It Down:

1. Reset Environment:
   - At the start of each episode, reset the environment to get an initial state.
2. Select Action:
   - Random Action: At the start of training, actions are random to encourage exploration.
   - Policy Action: Once the agent has learned a bit, it uses the Actor network to choose actions.
   - Noise is added for exploration.
3. Perform Action:
   - `env.step(action)`: Executes the action and returns:
     - `new_obs`: The next state.
     - `reward`: Feedback on the action's quality.
     - `done`: Whether the episode has ended.
4. Store Experience:
   - Save the `(state, action, reward, next state, done)` tuple in the `Replay Buffer`.
5. Update Counters:
   - Update `total_timesteps` and `episode_reward`.

Evaluation

Periodically, the agent is evaluated to measure its progress.

Code Reference:

python

```
def evaluate_policy(policy, eval_episodes=10):
      avg_reward = 0.
      for _ in range(eval_episodes):
            obs = env.reset()
            done = False
            while not done:
                  # Use Actor without noise
                  action = policy.select_action(np.array(obs))
                  obs, reward, done, _ = env.step(action)
                  avg_reward += reward
      avg_reward /= eval_episodes
      print(f"Average Reward: {avg_reward}")
      return avg_reward
```

Breaking It Down:

1. Evaluation Episodes:
   - The agent runs for a set number of episodes (eval_episodes) without exploration noise.
2. Calculate Average Reward:
   - Rewards are accumulated across all episodes.
   - Average reward shows how well the agent performs.

Key Concepts:

1. State and Action:

    - The agent observes the state and selects actions accordingly.
2. Feedback Loop:
    - The environment provides rewards and updates the state based on the agent’s actions.
3. Exploration vs. Exploitation:
    - Early in training, the agent tries random actions (exploration).
    - Later, it relies on its learned policy (exploitation).

Why This Matters

- Environment Interaction: Teaches the agent by trial and error.
- Evaluation: Measures how well the agent has learned.
- Exploration: Prevents the agent from getting stuck in bad habits early in training.

Step 7: Advanced Features in TD3

TD3 introduces three major innovations over standard DDPG to improve learning stability and performance. These advanced features include policy smoothing, delayed updates, and target networks.

---

1. Policy Smoothing

This feature prevents the Actor from overfitting to specific actions by adding random noise to the target actions during training.

Code Reference:

python

```
next_action = self.actor_target(next_state)
noise = torch.normal(0, policy_noise,
size=next_action.shape).clamp(-noise_clip, noise_clip).to(device)
```

```
next_action = (next_action + noise).clamp(-self.max_action, self.max_action)
```

Breaking It Down:

1. Target Action:
    - The target Actor (actor_target) predicts the next action for the next state.
2. Add Noise:
    - Gaussian noise (torch.normal) is added to the predicted action.
    - Noise is clamped between -noise_clip and +noise_clip to keep it reasonable.
3. Clip Action:
    - The final action is clamped between the allowable range (-max_action, +max_action).

Why It's Important:

- Helps avoid overfitting to Q-values by making the policy smoother.
- Prevents the Actor from relying too much on precise Q-value estimates.

---

2. Delayed Updates

The Actor network is updated less frequently than the Critics to allow the Critics to provide more accurate Q-values.

Code Reference:

python

```
if it % policy_freq == 0:
      # Update Actor
      actor_loss = -self.critic.Q1(state, self.actor(state)).mean()
      self.actor_optimizer.zero_grad()
```

```
actor_loss.backward()
self.actor_optimizer.step()

# Update target networks
for param, target_param in zip(self.actor.parameters(),
self.actor_target.parameters()):
      target_param.data.copy_(tau * param.data + (1 - tau) *
      target_param.data)
for param, target_param in zip(self.critic.parameters(),
self.critic_target.parameters()):
      target_param.data.copy_(tau * param.data + (1 - tau) *
      target_param.data)
```

Breaking It Down:

1. Frequency Check:
   - The Actor is updated only once every policy_freq iterations.
   - Example: If policy_freq = 2, the Actor is updated every second step.
2. Actor Update:
   - Uses the Critic's Q1 values to compute the Actor's loss.
   - The Actor learns to maximize the Q-value predicted by the first Critic.
3. Target Updates:
   - Both the Actor and Critic target networks are updated after the Actor is updated.
   - Uses Polyak Averaging to make updates gradual:

python

```
target_param.data.copy_(tau * param.data + (1 - tau) * target_param.data)
```

Why It's Important:

- Ensures the Actor doesn't update too quickly based on unstable Q-values.
- Gives the Critics time to learn more accurate estimates.

---

3. Target Networks

Target networks are slow-moving copies of the main Actor and Critic networks. They provide stable Q-value estimates during training.

Code Reference:

python

```python
self.actor_target.load_state_dict(self.actor.state_dict())
self.critic_target.load_state_dict(self.critic.state_dict())
```

How It Works:

1. Initialization:
    - At the start of training, the target networks are exact copies of the main networks.
2. Gradual Updates:
    - Target networks are updated slowly during training using Polyak Averaging:

python

```python
target_param.data.copy_(tau * param.data + (1 - tau) * target_param.data)
```

- tau controls the update rate (e.g., tau = 0.005 means very slow updates).

Why It's Important:

- Provides a stable baseline for computing target Q-values.
- Prevents training instability caused by rapidly changing networks.

How These Features Work Together

Let's summarize how these advanced features improve TD3:

1. Policy Smoothing:
    - Adds randomness to target actions to prevent overfitting.
2. Delayed Updates:
    - Ensures the Critic provides accurate Q-values before the Actor updates.
3. Target Networks:
    - Stabilize learning by providing consistent Q-value estimates.

Putting It All Together: Training Step Flow

Here's how a single training step works in TD3:

1. Sample Batch: Get a batch of experiences from the Replay Buffer.
2. Policy Smoothing: Add noise to the target action to calculate the target Q-value.
3. Critic Update: Update both Critics using the MSE loss between the predicted and target Q-values.
4. Delayed Actor Update: Update the Actor every policy_freq steps.
5. Target Network Update: Slowly update the target networks.

Why These Features Matter

- Stability: Reduces sudden shifts in learning.
- Robustness: Prevents the Actor from overfitting or relying too much on noisy Q-values.
- Efficiency: Focuses on improving actions that truly maximize rewards.

Step 8: Putting It All Together – TD3 Training Pipeline

Now that we've explored the key components and advanced features of TD3, let's look at how they come together in the overall training process. The pipeline involves interacting with the environment, storing experiences, training the networks, and periodically evaluating the agent's performance.

TD3 Training Pipeline

Here's the step-by-step flow of the TD3 training process

---

1. Initialize Components

Before training begins, all necessary components are initialized.

Code Reference

python

```
env_name = "AntBulletEnv-v0"
env = gym.make(env_name)

state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]
max_action = float(env.action_space.high[0])

policy = TD3(state_dim, action_dim, max_action)
replay_buffer = ReplayBuffer()
```

What Happens Here?:

1. Environment:
   - env_name specifies the task, such as making an ant walk.
   - state_dim and action_dim describe the input (state) and output (action) dimensions.
2. Policy:
   - The TD3 agent is created with the state and action dimensions.
3. Replay Buffer:

    ○ A memory buffer for storing experiences.

---

2. Training Loop

The agent interacts with the environment over many steps (e.g., 500,000 timesteps).

Code Reference:

python

```
total_timesteps = 0
done = True
while total_timesteps < max_timesteps:
      if done:
            obs = env.reset()
            done = False
            episode_reward = 0
            episode_timesteps = 0

            # Select action
            if total_timesteps < start_timesteps:
            action = env.action_space.sample()
            else:
            action = policy.select_action(np.array(obs))
            action = (action + np.random.normal(0, expl_noise,
            size=env.action_space.shape[0])).clip(env.action_space.low,
            env.action_space.high)

      # Take action in environment
      new_obs, reward, done, _ = env.step(action)

      # Store experience
      replay_buffer.add((obs, new_obs, action, reward, float(done)))
```

```
    # Train policy
    if total_timesteps >= start_timesteps:
          policy.train(replay_buffer, iterations=1, batch_size=100,
          discount=0.99, tau=0.005, policy_noise=0.2, noise_clip=0.5,
          policy_freq=2)

    obs = new_obs
    total_timesteps += 1
    episode_timesteps += 1
    episode_reward += reward
```

What Happens Here?:

1. Reset Environment:
   - If the episode ends, reset the environment and initialize counters.
2. Select Action:
   - Initially, random actions are taken to explore the environment.
   - After enough timesteps, the Actor network predicts actions, and small noise is added for exploration.
3. Take Action:
   - The environment executes the action and returns the next state (new_obs), reward, and whether the episode is done.
4. Store Experience:
   - Save (state, action, reward, next_state, done) in the Replay Buffer.
5. Train Policy:
   - Start training the networks once enough data has been collected.

---

3. Periodic Evaluation

Every few steps, the agent is evaluated to measure its performance.

Code Reference:

python

```
def evaluate_policy(policy, eval_episodes=10):
      avg_reward = 0.
      for _ in range(eval_episodes):
            obs = env.reset()
            done = False
            while not done:
                  action = policy.select_action(np.array(obs))
                  obs, reward, done, _ = env.step(action)
                  avg_reward += reward
      avg_reward /= eval_episodes
      print(f"Average Reward: {avg_reward}")
      return avg_reward
```

What Happens Here?:

1. Run Episodes:
   - The agent runs for several episodes without exploration noise.
2. Calculate Average Reward:
   - The rewards are averaged across all episodes to track progress.

---

4. Save and Load Models

The policy can be saved to disk and reloaded for later use.

Code Reference:

python

```
policy.save("TD3_model", "./models/")
policy.load("TD3_model", "./models/")
```

Why This Matters:

- Save: Store the trained model for deployment or further training.
- Load: Resume training or evaluate the policy without retraining from scratch.

Key Concepts in the Pipeline

1. Exploration vs. Exploitation:
    - Random actions early in training encourage the agent to explore the environment.
    - Later, the Actor's predictions are used to exploit what the agent has learned.
2. Batch Training:
    - The agent learns by sampling batches of experiences from the Replay Buffer.
3. Periodic Updates:
    - Delayed updates ensure stable learning.
4. Evaluation:
    - Helps monitor progress and adjust training if necessary.

TD3 Workflow Summary

Here's a high-level summary of the workflow:

1. Initialize the environment, agent, and Replay Buffer.
2. Run episodes, storing experiences in the Replay Buffer.
3. Train the policy by sampling from the Replay Buffer.
4. Periodically evaluate the agent to track its performance.
5. Save the policy for future use.

Step 9: Applying TD3 to Specific Environments

Now that we understand the TD3 pipeline, let's focus on how TD3 is applied to specific environments, such as Half-Cheetah, Ant, and Humanoid. Each environment presents unique challenges that affect how the agent is trained and evaluated.

Environment Overview

The environments are continuous control tasks where the agent learns to move effectively:

1. Half-Cheetah: A 2D robot that learns to run forward by controlling its joints.
2. Ant: A 3D robot with four legs that learns to walk.
3. Humanoid: A more complex 3D robot that learns to move upright.

Each environment has:

- Observation Space: The agent's inputs (e.g., joint angles, velocities).
- Action Space: The agent's outputs (e.g., torques applied to joints).
- Reward Function: A measure of success, often based on how far the agent moves forward.

Customizing TD3 for Half-Cheetah

In the Half-Cheetah environment, the agent needs to control a simple robot with a few joints. This makes it a good starting point for TD3.

Code Reference:

python

```
env_name = "HalfCheetah-v2"
env = gym.make(env_name)

state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]
max_action = float(env.action_space.high[0])

policy = TD3(state_dim, action_dim, max_action)
```

Key Adjustments:

1. Environment Name:
   - Specify HalfCheetah-v2 to load the Half-Cheetah task.
2. State and Action Dimensions:
   - `state_dim`: Number of observations (e.g., joint angles and velocities).
   - `action_dim`: Number of actions (e.g., torques applied to joints).

Challenges in Half-Cheetah

1. Balancing Speed and Stability:
   - The agent must learn to run fast without falling.
2. Reward Signal:
   - Rewards are based on forward velocity.

Customizing TD3 for Ant

The Ant environment adds complexity with 3D movement and more joints.

Code Reference:

python

```
env_name = "AntBulletEnv-v0"
env = gym.make(env_name)

state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]
max_action = float(env.action_space.high[0])

policy = TD3(state_dim, action_dim, max_action)
```

Key Adjustments:

1. Environment Name:
   - Use `AntBulletEnv-v0` for the Ant task.
2. Observation Space:
   - Includes additional dimensions for 3D movement.
3. Action Space:
   - Controls more joints than Half-Cheetah.

Challenges in Ant

1. Complex Movement:
   - The agent must learn to coordinate all four legs.
2. 3D Environment:
   - Adds balance as a critical factor.

Customizing TD3 for Humanoid

The Humanoid environment is the most challenging, requiring the agent to control a humanoid robot in 3D space.

Code Reference:

python

```python
env_name = "HumanoidBulletEnv-v0"
env = gym.make(env_name)

state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]
max_action = float(env.action_space.high[0])

policy = TD3(state_dim, action_dim, max_action)
```

Key Adjustments:

1. Environment Name:
   - Specify `HumanoidBulletEnv-v0`.
2. State and Action Dimensions:
   - Larger state and action spaces compared to Ant or Half-Cheetah.

Challenges in Humanoid

1. High Degrees of Freedom:
   - More joints and actions make training slower and more complex.
2. Balance:
   - The agent must learn to stay upright while moving efficiently.

Key Differences Across Environments

Environment	State Dim	Action Dim	Complexity	Reward Signal
Half-Cheetah	Small	Small	Low	Forward velocity
Ant	Medium	Medium	Moderate	Balance and forward movement
Humanoid	Large	Large	High	Staying upright and moving forward

Training Customization

For each environment:

1. Tuning Parameters:
   - Exploration Noise (`expl_noise`): Adjust for environments with more complex actions.
   - Policy Noise (`policy_noise`): Smaller noise helps in delicate environments like Humanoid.
2. Reward Scaling:

    - Rewards might need scaling for environments with larger state or action spaces.
3. Evaluation Frequency:
    - Complex environments require longer training, so evaluation may be less frequent.

Summary

- Half-Cheetah: Ideal for starting TD3 training due to simplicity.
- Ant: Introduces challenges like 3D movement and balance.
- Humanoid: Requires advanced coordination and balance, making it the most complex task.
