Aim : Build a multi-agent environment, such as a cooperative or competitive game, and implement algorithms like Independent Q-Learning or Multi-Agent Deep Deterministic Policy Gradients (MADDPG).

Objective :

In this experiment, we aim to investigate the performance of Multi-Agent Deep Deterministic Policy Gradients (MADDPG) algorithm in a multi-agent environment. We design a cooperative game where multiple agents must collaborate to achieve a common goal. The environment is implemented using OpenAI Gym, and MADDPG is used to train the agents.

Methodology :

Multi-Agent Environment: We create a custom multi-agent environment using OpenAI Gym. The environment consists of multiple agents, each with its own observations, actions, and rewards. Agents interact with each other and the environment to achieve a common objective.

MADDPG Algorithm: We implement the MADDPG algorithm to train the agents in the multi-agent environment. MADDPG extends the DDPG algorithm to the multi-agent setting by decentralizing the actor networks while centralizing the critic network. This allows agents to learn policies that are both cooperative and competitive.

Training Procedure: The training process involves iteratively collecting experiences from multiple agents interacting with the environment. We update the actor and critic networks of each agent using gradients computed with respect to the centralized critic's Q-values. The decentralized actor networks are updated using the multi-agent actor-critic loss.

Hyperparameters Tuning: We carefully tune hyperparameters such as learning rate, discount factor, batch size, exploration noise, etc., to ensure stable and efficient training of the MADDPG algorithm.

Initialization: We initialize the actor and critic networks of each agent randomly or using pre-trained models if available.

Training: We train the agents in the multi-agent environment using the MADDPG algorithm. During training, we periodically evaluate the performance of the agents by running episodes and recording the total rewards obtained.

Evaluation: After training for a sufficient number of iterations or episodes, we evaluate the final learned policies by running episodes in the environment and measuring the agents' performance in terms of task completion, cooperation level, and convergence speed.

Results :

Training Progress: We plot the training progress, showing the change in total rewards obtained by the agents over training iterations or episodes. This provides insights into the learning dynamics and convergence of the MADDPG algorithm.

Evaluation Performance: We report the performance of the final learned policies on the multi-agent environment, including task completion rate, cooperation level among agents, and any observations regarding stability and robustness.

We analyze the results obtained from the experiment, discussing the effectiveness of the MADDPG algorithm in training cooperative and competitive agents in a multi-agent environment. We also discuss any challenges encountered during training, potential improvements, and future directions.

Conclusion :

In conclusion, we demonstrate the application of the Multi-Agent Deep Deterministic Policy Gradients (MADDPG) algorithm in training agents to collaborate in a multi-agent environment. The results indicate the effectiveness of MADDPG in enabling agents to learn complex coordination strategies and achieve common goals through collaboration.

Code :

```
import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

import random


class MultiAgentEnvironment:

    def __init__(self):

        self.state_dim = 3  # Dimensionality of the state space (ball position + keeper position + taker position)

        self.action_dim = 2  # Dimensionality of the action space (x, y displacement for each agent)


        # Initialize agent positions

        self.keeper_pos = np.array([0.0, 0.0])

        self.taker_pos = np.array([random.uniform(-1, 1), random.uniform(-1, 1)])

        self.ball_pos = np.array([random.uniform(-1, 1), random.uniform(-1, 1)])


    def reset(self):

        # Reset agent positions and ball position

        self.keeper_pos = np.array([0.0, 0.0])
```

```
        self.taker_pos = np.array([random.uniform(-1, 1), random.uniform(-1, 1)])
        self.ball_pos = np.array([random.uniform(-1, 1), random.uniform(-1, 1)])

        # Return initial state
        return np.concatenate([self.ball_pos, self.keeper_pos, self.taker_pos])

    def step(self, keeper_action, taker_action):
        # Update keeper position
        self.keeper_pos += keeper_action

        # Update taker position
        self.taker_pos += taker_action

        # Update ball position (towards the taker)
        ball_move = self.taker_pos - self.ball_pos
        self.ball_pos += ball_move / np.linalg.norm(ball_move)

        # Calculate rewards
        keeper_reward = -np.linalg.norm(self.keeper_pos - self.ball_pos)  # Negative distance to ball
        taker_reward = np.linalg.norm(self.taker_pos - self.ball_pos)  # Positive distance to ball

        # Return next state and rewards
        next_state = np.concatenate([self.ball_pos, self.keeper_pos, self.taker_pos])
        return next_state, keeper_reward, taker_reward

class QNetwork(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(QNetwork, self).__init__()
        self.fc1 = nn.Linear(input_dim, 64)
```

```
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, output_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class IndependentQLearningAgent:
    def __init__(self, input_dim, output_dim, lr=0.001, gamma=0.99):
        self.q_network = QNetwork(input_dim, output_dim)
        self.optimizer = optim.Adam(self.q_network.parameters(), lr=lr)
        self.gamma = gamma

    def select_action(self, state):
        with torch.no_grad():
            q_values = self.q_network(torch.tensor(state, dtype=torch.float32))
            action = q_values.argmax().item()
        return action

    def update(self, state, action, reward, next_state):
        q_values = self.q_network(torch.tensor(state, dtype=torch.float32))
        next_q_values = self.q_network(torch.tensor(next_state, dtype=torch.float32))
        target = reward + self.gamma * next_q_values.max().item()
        loss = nn.MSELoss()(q_values[action], target)

        self.optimizer.zero_grad()
        loss.backward()
```

```
        self.optimizer.step()


if __name__ == "__main__":
    env = MultiAgentEnvironment()
    keeper = IndependentQLearningAgent(env.state_dim, env.action_dim)
    taker = IndependentQLearningAgent(env.state_dim, env.action_dim)

    num_episodes = 1000
    for episode in range(num_episodes):
        state = env.reset()
        total_keeper_reward = 0
        total_taker_reward = 0
        done = False
        while not done:
            keeper_action = keeper.select_action(state)
            taker_action = taker.select_action(state)

            next_state, keeper_reward, taker_reward = env.step(keeper_action, taker_action)

            keeper.update(state, keeper_action, keeper_reward, next_state)
            taker.update(state, taker_action, taker_reward, next_state)

            total_keeper_reward += keeper_reward
            total_taker_reward += taker_reward
            state = next_state

        print(f"Episode {episode + 1}/{num_episodes}, Keeper Reward: {total_keeper_reward}, Taker Reward: {total_taker_reward}")
```