Custom Multi-Agent RL Environment with PyTorch and Gymnasium

Below is a complete implementation of a custom multi-agent environment using Gymnasium and PyTorch, with support for both cooperative and competitive scenarios.

1. Environment Setup

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.distributions import Categorical
from gymnasium import spaces
from gymnasium.core import Env
import matplotlib.pyplot as plt
from matplotlib.patches import Circle, Rectangle
from collections import defaultdict, deque
import random
```

2. Custom Multi-Agent Environment Class

```
class MultiAgentGridWorld(Env):
    """Custom multi-agent grid world environment"""

    metadata = {'render_modes': ['human', 'rgb_array'], 'render_fps': 4}

    def __init__(self,
                 grid_size=10,
                 num_agents=2,
                 num_targets=2,
                 mode='cooperative',
                 max_steps=100,
                 render_mode=None):
        """
        Args:
            grid_size: Size of the square grid
            num_agents: Number of agents in the environment
            num_targets: Number of targets to collect
            mode: 'cooperative' or 'competitive'
            max_steps: Maximum steps per episode
            render_mode: 'human' or 'rgb_array'
        """
        super().__init__()

        self.grid_size = grid_size
```

PROF

```
        self.num_agents = num_agents
        self.num_targets = num_targets
        self.mode = mode
        self.max_steps = max_steps
        self.render_mode = render_mode

        # Action space: 0=up, 1=down, 2=left, 3=right, 4=stay
        self.action_space = spaces.Discrete(5)

        # Observation space: grid position + other agents' positions +
target positions
        self.observation_space = spaces.Dict({
            'agent_pos': spaces.Box(low=0, high=grid_size-1, shape=(2,),
dtype=int),
            'other_pos': spaces.Box(low=0, high=grid_size-1,
                                   shape=(num_agents-1, 2), dtype=int),
            'target_pos': spaces.Box(low=0, high=grid_size-1,
                                   shape=(num_targets, 2), dtype=int),
            'target_status': spaces.MultiBinary(num_targets)
        })

        # Colors for visualization
        self.agent_colors = plt.cm.tab10(np.linspace(0, 1, num_agents))
        self.target_color = np.array([0.8, 0.2, 0.2, 1])

        # Initialize state
        self.reset()

    def reset(self, seed=None, options=None):
        super().reset(seed=seed)

        # Initialize agent positions (non-overlapping)
        self.agent_positions = []
        while len(self.agent_positions) < self.num_agents:
            pos = (self.np_random.integers(0, self.grid_size, size=2))
            if pos not in self.agent_positions:
                self.agent_positions.append(pos)

        # Initialize target positions and status
        self.target_positions = []
        while len(self.target_positions) < self.num_targets:
            pos = (self.np_random.integers(0, self.grid_size, size=2))
            if pos not in self.agent_positions and pos not in
self.target_positions:
                self.target_positions.append(pos)
        self.target_status = np.ones(self.num_targets, dtype=bool)

        self.steps = 0
        self.agents = [f"agent_{i}" for i in range(self.num_agents)]

        # For rendering
        if self.render_mode == 'human':
            self._setup_render()
```

PROF

```
        return self._get_obs(), self._get_info()

    def _get_obs(self):
        """Return observations for all agents"""
        observations = {}

        for i, agent in enumerate(self.agents):
            other_pos = []
            for j, pos in enumerate(self.agent_positions):
                if j != i:
                    other_pos.append(pos)

            observations[agent] = {
                'agent_pos': np.array(self.agent_positions[i]),
                'other_pos': np.array(other_pos),
                'target_pos': np.array(self.target_positions),
                'target_status': np.array(self.target_status.copy())
            }

        return observations

    def _get_info(self):
        """Return additional info (not used for learning)"""
        return {
            'agent_positions': self.agent_positions.copy(),
            'target_positions': self.target_positions.copy(),
            'target_status': self.target_status.copy(),
            'steps': self.steps
        }

    def step(self, actions):
        """Execute one time step in the environment"""

        rewards = {agent: 0 for agent in self.agents}
        terminated = {agent: False for agent in self.agents}
        truncated = {agent: False for agent in self.agents}
        self.steps += 1

        # Move agents
        for i, (agent, action) in enumerate(actions.items()):
            if action == 0:  # Up
                self.agent_positions[i][1] = min(self.grid_size-1,
self.agent_positions[i][1] + 1)
            elif action == 1:  # Down
                self.agent_positions[i][1] = max(0,
self.agent_positions[i][1] - 1)
            elif action == 2:  # Left
                self.agent_positions[i][0] = max(0,
self.agent_positions[i][0] - 1)
            elif action == 3:  # Right
                self.agent_positions[i][0] = min(self.grid_size-1,
self.agent_positions[i][0] + 1)
```

PROF

```
            # Action 4 is stay (no movement)

        # Check target collection
        for t in range(self.num_targets):
            if self.target_status[t]:  # If target is active
                for i, agent in enumerate(self.agents):
                    if np.array_equal(self.agent_positions[i],
self.target_positions[t]):
                        if self.mode == 'cooperative':
                            rewards[agent] += 10  # Shared reward
                        else:
                            rewards[agent] += 20  # Individual reward
                            for other in self.agents:
                                if other != agent:
                                    rewards[other] -= 5  # Penalty for
others
                        self.target_status[t] = False  # Collect target

        # Time limit
        if self.steps >= self.max_steps:
            truncated = {agent: True for agent in self.agents}

        # All targets collected
        if not any(self.target_status):
            terminated = {agent: True for agent in self.agents}
            if self.mode == 'cooperative':
                # Additional completion bonus
                for agent in self.agents:
                    rewards[agent] += 20

        # Small step penalty
        for agent in self.agents:
            rewards[agent] -= 0.1

        return self._get_obs(), rewards, terminated, truncated,
self._get_info()

    def render(self):
        """Render the environment"""
        if self.render_mode is None:
            return

        if not hasattr(self, 'fig'):
            self._setup_render()

        # Clear the previous render
        self.ax.clear()

        # Draw grid
        for x in range(self.grid_size + 1):
            self.ax.axhline(x, color='gray', lw=1)
            self.ax.axvline(x, color='gray', lw=1)
```

```
        # Draw targets
        for t, pos in enumerate(self.target_positions):
            if self.target_status[t]:
                target = Circle((pos[0] + 0.5, pos[1] + 0.5), 0.4,
                              color=self.target_color)
                self.ax.add_patch(target)
                self.ax.text(pos[0] + 0.5, pos[1] + 0.5, str(t),
                            ha='center', va='center', color='white')

        # Draw agents
        for i, pos in enumerate(self.agent_positions):
            agent = Circle((pos[0] + 0.5, pos[1] + 0.5), 0.3,
                          color=self.agent_colors[i])
            self.ax.add_patch(agent)
            self.ax.text(pos[0] + 0.5, pos[1] + 0.5, str(i),
                        ha='center', va='center', color='white')

        # Set plot limits and labels
        self.ax.set_xlim(0, self.grid_size)
        self.ax.set_ylim(0, self.grid_size)
        self.ax.set_xticks(np.arange(0, self.grid_size + 1, 1))
        self.ax.set_yticks(np.arange(0, self.grid_size + 1, 1))
        self.ax.set_title(f'Multi-Agent Grid World (Step:
{self.steps})')
        self.ax.grid(True)

        if self.render_mode == 'human':
            plt.pause(0.1)
        elif self.render_mode == 'rgb_array':
            self.fig.canvas.draw()
            img = np.frombuffer(self.fig.canvas.tostring_rgb(),
dtype=np.uint8)
            img = img.reshape(self.fig.canvas.get_width_height()[::-1] +
(3,))
            return img

    def _setup_render(self):
        """Initialize rendering components"""
        if self.render_mode == 'human':
            plt.ion()
            self.fig, self.ax = plt.subplots(figsize=(8, 8))
        elif self.render_mode == 'rgb_array':
            self.fig, self.ax = plt.subplots(figsize=(8, 8))

    def close(self):
        """Close the environment and any rendering windows"""
        if hasattr(self, 'fig'):
            plt.close(self.fig)
            plt.ioff()
```


3. Multi-Agent Policy Network

```
class MultiAgentPolicy(nn.Module):
    """Policy network shared by all agents"""

    def __init__(self, obs_space, action_space, hidden_size=128):
        super().__init__()

        # Calculate input size from observation space
        self.input_size = (
            obs_space['agent_pos'].shape[0] +
            obs_space['other_pos'].shape[0] *
obs_space['other_pos'].shape[1] +
            obs_space['target_pos'].shape[0] *
obs_space['target_pos'].shape[1] +
            obs_space['target_status'].shape[0]
        )

        self.action_size = action_space.n

        # Shared feature extractor
        self.shared_net = nn.Sequential(
            nn.Linear(self.input_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, hidden_size),
            nn.ReLU()
        )

        # Policy head
        self.policy_head = nn.Linear(hidden_size, self.action_size)

        # Value head
        self.value_head = nn.Linear(hidden_size, 1)

    def forward(self, obs):
        # Flatten observations
        x = torch.cat([
            obs['agent_pos'].float(),
            obs['other_pos'].flatten().float(),
            obs['target_pos'].flatten().float(),
            obs['target_status'].float()
        ], dim=-1)

        # Shared features
        features = self.shared_net(x)

        # Policy logits
        logits = self.policy_head(features)

        # Value estimate
        value = self.value_head(features)

        return logits, value
```

PROF

4. PPO Implementation for Multi-Agent

```
class MultiAgentPPO:
    """PPO implementation for multi-agent setting"""

    def __init__(self, env, lr=3e-4, gamma=0.99, epsilon=0.2,
                 entropy_coef=0.01, clip_grad=0.5, update_epochs=4):
        self.env = env
        self.gamma = gamma
        self.epsilon = epsilon
        self.entropy_coef = entropy_coef
        self.clip_grad = clip_grad
        self.update_epochs = update_epochs

        # Initialize policy
        self.policy = MultiAgentPolicy(
            env.observation_space,
            env.action_space
        )

        self.optimizer = optim.Adam(self.policy.parameters(), lr=lr)
        self.memory = defaultdict(list)

    def act(self, obs):
        """Get actions for all agents"""
        actions = {}
        log_probs = {}
        values = {}

        for agent, agent_obs in obs.items():
            # Convert observation to tensor
            agent_obs = {
                k: torch.from_numpy(v).unsqueeze(0)
                for k, v in agent_obs.items()
            }

            # Get action distribution and value
            logits, value = self.policy(agent_obs)
            dist = Categorical(logits=logits)
            action = dist.sample()

            actions[agent] = action.item()
            log_probs[agent] = dist.log_prob(action).item()
            values[agent] = value.item()

        return actions, log_probs, values

    def store_experience(self, obs, actions, log_probs, values, rewards,
dones):
        """Store experience in memory"""
```

PROF

```
        for agent in self.env.agents:
            self.memory[agent].append((
                obs[agent],
                actions[agent],
                log_probs[agent],
                values[agent],
                rewards[agent],
                dones[agent]
            ))

    def compute_returns(self, rewards, dones, last_value):
        """Compute discounted returns"""
        returns = []
        R = last_value
        for step in reversed(range(len(rewards))):
            R = rewards[step] + self.gamma * R * (1 - dones[step])
            returns.insert(0, R)
        return returns

    def update(self):
        """Update policy using PPO"""
        if not all(len(mem) > 0 for mem in self.memory.values()):
            return

        # Process each agent's experience
        policy_loss = 0
        value_loss = 0
        entropy_loss = 0

        for agent in self.env.agents:
            # Unpack memory
            obs, actions, old_log_probs, old_values, rewards, dones =
zip(*self.memory[agent])

            # Convert to tensors
            obs = {
                k: torch.FloatTensor(np.array([o[k] for o in obs]))
                for k in obs[0].keys()
            }
            actions = torch.LongTensor(actions)
            old_log_probs = torch.FloatTensor(old_log_probs)
            old_values = torch.FloatTensor(old_values)
            rewards = torch.FloatTensor(rewards)
            dones = torch.FloatTensor(dones)

            # Compute returns and advantages
            returns = torch.FloatTensor(self.compute_returns(rewards,
dones, old_values[-1]))
            advantages = returns - old_values
            advantages = (advantages - advantages.mean()) /
(advantages.std() + 1e-8)

            # Optimize for several epochs
```

```
            for _ in range(self.update_epochs):
                # Get new logits and values
                logits, new_values = self.policy(obs)
                dist = Categorical(logits=logits)

                # New log probs and entropy
                new_log_probs = dist.log_prob(actions)
                entropy = dist.entropy().mean()

                # Ratio between new and old policies
                ratio = (new_log_probs - old_log_probs).exp()

                # Clipped surrogate objective
                surr1 = ratio * advantages
                surr2 = torch.clamp(ratio, 1.0 - self.epsilon, 1.0 +
self.epsilon) * advantages
                actor_loss = -torch.min(surr1, surr2).mean()

                # Value function loss
                critic_loss = F.mse_loss(new_values.squeeze(), returns)

                # Accumulate losses
                policy_loss += actor_loss
                value_loss += critic_loss
                entropy_loss += entropy

        # Normalize by number of agents
        num_agents = len(self.env.agents)
        policy_loss /= num_agents
        value_loss /= num_agents
        entropy_loss /= num_agents

        # Total loss
        loss = policy_loss + 0.5 * value_loss - self.entropy_coef *
entropy_loss

        # Gradient step
        self.optimizer.zero_grad()
        loss.backward()
        nn.utils.clip_grad_norm_(self.policy.parameters(),
self.clip_grad)
        self.optimizer.step()

        # Clear memory
        self.memory.clear()

        return {
            'policy_loss': policy_loss.item(),
            'value_loss': value_loss.item(),
            'entropy': entropy_loss.item()
        }

    def save(self, filename):
```

```
        torch.save(self.policy.state_dict(), filename)

    def load(self, filename):
        self.policy.load_state_dict(torch.load(filename))
```

5. Training Loop

```
def train_multi_agent_ppo():
    # Create environment
    env = MultiAgentGridWorld(
        grid_size=8,
        num_agents=2,
        num_targets=3,
        mode='cooperative',  # Try 'competitive' for different behavior
        max_steps=200,
        render_mode=None  # Change to 'human' to visualize training
    )

    # Initialize PPO
    ppo = MultiAgentPPO(env)

    # Training parameters
    episodes = 1000
    batch_size = 1024  # Total steps across all agents before update

    # Tracking
    episode_rewards = []
    avg_rewards = []
    losses = []

    for ep in range(1, episodes + 1):
        obs, _ = env.reset()
        total_rewards = {agent: 0 for agent in env.agents}
        steps = 0

        while True:
            # Get actions
            actions, log_probs, values = ppo.act(obs)

            # Step environment
            next_obs, rewards, terminated, truncated, _ =
env.step(actions)

            # Check if all agents are done
            done = all(terminated.values()) or all(truncated.values())

            # Store experience
            ppo.store_experience(obs, actions, log_probs, values,
rewards, terminated)
```

```
            # Update total rewards
            for agent in env.agents:
                total_rewards[agent] += rewards[agent]

            # Update policy if we have enough samples
            if steps % batch_size == 0 or done:
                loss_info = ppo.update()
                if loss_info:
                    losses.append(loss_info)

            # Next step
            obs = next_obs
            steps += 1

            if done:
                break

        # Track rewards
        mean_ep_reward = sum(total_rewards.values()) /
len(total_rewards)
        episode_rewards.append(mean_ep_reward)
        avg_rewards.append(np.mean(episode_rewards[-100:]))

        # Print progress
        print(f"Episode {ep}, Reward: {mean_ep_reward:.1f}, Avg Reward:
{avg_rewards[-1]:.1f}")

        # Early stopping if solved
        if avg_rewards[-1] >= 100:  # Adjust threshold based on your
environment
            print(f"Solved in {ep} episodes!")
            ppo.save('multi_agent_ppo.pth')
            break

    # Plot results
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    plt.plot(episode_rewards, alpha=0.5, label='Episode Reward')
    plt.plot(avg_rewards, label='100-Episode Avg')
    plt.xlabel('Episode')
    plt.ylabel('Reward')
    plt.legend()
    plt.grid(True)

    if losses:
        plt.subplot(1, 2, 2)
        plt.plot([l['policy_loss'] for l in losses], label='Policy
Loss')
        plt.plot([l['value_loss'] for l in losses], label='Value Loss')
        plt.plot([l['entropy'] for l in losses], label='Entropy')
        plt.xlabel('Update Step')
        plt.ylabel('Loss')
```

```
        plt.legend()
        plt.grid(True)

    plt.tight_layout()
    plt.show()

    env.close()

if __name__ == "__main__":
    train_multi_agent_ppo()
```

6. Evaluation

```
def evaluate_multi_agent_ppo(episodes=5):
    # Create environment with rendering
    env = MultiAgentGridWorld(
        grid_size=8,
        num_agents=2,
        num_targets=3,
        mode='cooperative',
        max_steps=200,
        render_mode='human'
    )

    # Initialize PPO and load trained weights
    ppo = MultiAgentPPO(env)
    ppo.load('multi_agent_ppo.pth')

    for ep in range(1, episodes + 1):
        obs, _ = env.reset()
        total_rewards = {agent: 0 for agent in env.agents}
        done = False

        while not done:
            # Get actions (no exploration during evaluation)
            actions, _, _ = ppo.act(obs)

            # Step environment
            obs, rewards, terminated, truncated, _ = env.step(actions)

            # Update total rewards
            for agent in env.agents:
                total_rewards[agent] += rewards[agent]

            # Check if all agents are done
            done = all(terminated.values()) or all(truncated.values())

        print(f"Evaluation Episode {ep}, Rewards: {total_rewards}")

    env.close()
```

PROF

```
# Uncomment to evaluate
# evaluate_multi_agent_ppo()
```

Key Features:

1. **Custom Multi-Agent Environment**:
    - Grid world with multiple agents and targets
    - Supports both cooperative and competitive modes
    - Configurable grid size, agent count, and target count
    - Visualization with matplotlib
2. **Shared Policy Network**:
    - Single network processes observations for all agents
    - Handles variable number of agents
    - Produces both policy logits and value estimates
3. **PPO Implementation**:
    - Clipped objective for stable policy updates
    - Advantage normalization
    - Entropy bonus for exploration
    - Gradient clipping
4. **Training Infrastructure**:
    - Experience collection from multiple agents
    - Batch updates with multiple epochs
    - Reward tracking and visualization

This implementation provides a solid foundation for experimenting with multi-agent RL. You can extend it by:

- Adding communication between agents
- Implementing different reward structures
- Trying more complex environments
- Experimenting with hierarchical policies
- Adding opponent modeling in competitive scenarios