Multi-Agent Environment Simulation with RLlib (PyTorch)

Below is a comprehensive example of setting up a multi-agent environment with RLlib for both cooperative and competitive tasks using PyTorch.

1. Setting Up a Custom Multi-Agent Environment

First, let's create a custom environment that supports both cooperative and competitive scenarios:

```
import numpy as np
import gym
from gym.spaces import Discrete, Box
from ray.rllib.env.multi_agent_env import MultiAgentEnv

class MultiAgentGridWorld(MultiAgentEnv):
    """
    A grid world where agents can cooperate or compete based on reward structure.
    """
    def __init__(self, config=None):
        config = config or {}
        self.size = config.get("grid_size", 5)
        self.max_steps = config.get("max_steps", 100)
        self.cooperative = config.get("cooperative", True)
        self.num_agents = config.get("num_agents", 2)

        # Define observation and action spaces
        self.observation_space = Box(low=0, high=1, shape=(self.size*self.size + 2*self.num_agents,))
        self.action_space = Discrete(4)  # Up, Down, Left, Right

        # Agents IDs
        self.agents = [f"agent_{i}" for i in range(self.num_agents)]

        # Environment state
        self.reset()

    def reset(self):
        self.steps = 0
        self.agent_positions = {
            agent: np.random.randint(0, self.size, size=2)
            for agent in self.agents
        }

        # Place targets
        if self.cooperative:
            # Shared target for cooperation
```

```
            self.target = np.random.randint(0, self.size, size=2)
        else:
            # Separate targets for competition
            self.targets = {
                agent: np.random.randint(0, self.size, size=2)
                for agent in self.agents
            }

        return self._get_obs()

    def _get_obs(self):
        obs = {}
        grid = np.zeros((self.size, self.size))

        for agent in self.agents:
            # Agent's own position
            pos = self.agent_positions[agent]
            grid[pos[0], pos[1]] = 1

            # Other agents' positions
            other_agents = [a for a in self.agents if a != agent]
            for i, other in enumerate(other_agents):
                opos = self.agent_positions[other]
                grid[opos[0], opos[1]] = 0.5

            # Target information
            if self.cooperative:
                grid[self.target[0], self.target[1]] = -1
                target_info = self.target
            else:
                grid[self.targets[agent][0], self.targets[agent][1]] = -1

                target_info = self.targets[agent]

            # Flatten grid and add agent-specific info
            agent_obs = np.concatenate([
                grid.flatten(),
                pos,
                target_info
            ])
            obs[agent] = agent_obs

        return obs

    def step(self, actions):
        rewards = {agent: 0 for agent in self.agents}
        dones = {"__all__": False}
        self.steps += 1

        # Move agents
        for agent, action in actions.items():
            pos = self.agent_positions[agent]
```

```
            # Action effects
            if action == 0:  # Up
                pos[0] = max(0, pos[0] - 1)
            elif action == 1:  # Down
                pos[0] = min(self.size - 1, pos[0] + 1)
            elif action == 2:  # Left
                pos[1] = max(0, pos[1] - 1)
            elif action == 3:  # Right
                pos[1] = min(self.size - 1, pos[1] + 1)

            # Check if reached target
            if self.cooperative:
                if np.array_equal(pos, self.target):
                    rewards[agent] = 1
                    if all(np.array_equal(self.agent_positions[a],
self.target) for a in self.agents):
                        rewards = {a: 10 for a in self.agents}  # Big
reward for all if all reach target
                        dones["__all__"] = True
            else:
                if np.array_equal(pos, self.targets[agent]):
                    rewards[agent] = 10
                    # Negative reward for others in competitive mode
                    for other in self.agents:
                        if other != agent:
                            rewards[other] = -5
                    dones["__all__"] = True

        # Timeout
        if self.steps >= self.max_steps:
            dones["__all__"] = True

        obs = self._get_obs()
        info = {}  # Additional info if needed

        return obs, rewards, dones, info
```


2. Configuring and Training with RLlib

Now let's set up the RLlib training configuration for both cooperative and competitive scenarios:

```
import ray
from ray import tune
from ray.rllib.agents.ppo import PPOTorchTrainer
from ray.rllib.models import ModelCatalog
from ray.rllib.models.torch.torch_modelv2 import TorchModelV2
import torch
import torch.nn as nn

# Define a custom neural network model
```

```
class CustomModel(TorchModelV2, nn.Module):
    def __init__(self, obs_space, action_space, num_outputs,
model_config, name):
        TorchModelV2.__init__(self, obs_space, action_space,
num_outputs, model_config, name)
        nn.Module.__init__(self)

        self.fcnet = nn.Sequential(
            nn.Linear(obs_space.shape[0], 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
        )

        self.action_out = nn.Linear(64, num_outputs)
        self.value_out = nn.Linear(64, 1)

    def forward(self, input_dict, state, seq_lens):
        features = self.fcnet(input_dict["obs"])
        self._value_out = self.value_out(features)
        return self.action_out(features), state

    def value_function(self):
        return self._value_out.squeeze(1)

# Register the custom model
ModelCatalog.register_custom_model("custom_model", CustomModel)

def train_multi_agent(config, coop=True):
    # Initialize Ray
    ray.init(ignore_reinit_error=True)

    # Configuration
    config = {
        "env": MultiAgentGridWorld,
        "env_config": {
            "grid_size": 5,
            "num_agents": 2,
            "cooperative": coop,
            "max_steps": 100,
        },
        "multiagent": {
            "policies": {
                # Define one policy per agent (could share or have
separate policies)
                f"policy_{i}": (
                    None,  # Use default obs/act spaces from env
                    Box(0, 1, (5*5 + 2*2,)),  # Custom obs space
                    Discrete(4),  # Action space
                    {
                        "model": {
                            "custom_model": "custom_model",
                        },
```

```
                        "gamma": 0.95,
                    }
                ) for i in range(2)
            },
            "policy_mapping_fn": lambda agent_id:
f"policy_{int(agent_id.split('_')[1])}",
        },
        "framework": "torch",
        "num_workers": 3,
        "num_envs_per_worker": 5,
        "train_batch_size": 4000,
        "rollout_fragment_length": 200,
        "sgd_minibatch_size": 256,
        "lr": 1e-4,
    }

    # Select algorithm (PPO in this case)
    trainer = PPOTorchTrainer(config=config)

    # Training loop
    for i in range(10):  # 10 training iterations
        result = trainer.train()
        print(f"Iteration {i}:")
        print(f"  - Total reward: {result['episode_reward_mean']}")
        print(f"  - Episode length: {result['episode_len_mean']}")

        # Optionally save the model
        if i % 5 == 0:
            checkpoint = trainer.save()
            print(f"Checkpoint saved at {checkpoint}")

    # Shutdown Ray when done
    ray.shutdown()

# Train cooperative agents
print("Training cooperative agents...")
train_multi_agent(coop=True)

# Train competitive agents
print("\nTraining competitive agents...")
train_multi_agent(coop=False)
```


3. Evaluating the Trained Agents

After training, you can evaluate the agents:

```
def evaluate_agents(coop=True):
    # Initialize Ray
    ray.init(ignore_reinit_error=True)
```

```python
    # Create environment
    env = MultiAgentGridWorld({
        "grid_size": 5,
        "num_agents": 2,
        "cooperative": coop,
        "max_steps": 100,
    })

    # Load trained policies
    if coop:
        checkpoint_path = "path_to_cooperative_checkpoint"
    else:
        checkpoint_path = "path_to_competitive_checkpoint"

    trainer = PPOTorchTrainer(config={
        "env": MultiAgentGridWorld,
        "framework": "torch",
        "num_workers": 0,
    })
    trainer.restore(checkpoint_path)

    # Run evaluation episodes
    for ep in range(3):  # Run 3 evaluation episodes
        obs = env.reset()
        done = {"__all__": False}
        total_rewards = {agent: 0 for agent in env.agents}

        while not done["__all__"]:
            actions = {}
            for agent_id in obs.keys():
                policy_id = f"policy_{int(agent_id.split('_')[1])}"
                actions[agent_id] =
trainer.compute_action(obs[agent_id], policy_id=policy_id)

            obs, rewards, done, _ = env.step(actions)

            for agent_id in rewards:
                total_rewards[agent_id] += rewards[agent_id]

        print(f"Episode {ep + 1} rewards:")
        for agent_id, reward in total_rewards.items():
            print(f"  {agent_id}: {reward}")

    ray.shutdown()

# Evaluate cooperative agents
print("Evaluating cooperative agents...")
evaluate_agents(coop=True)

# Evaluate competitive agents
print("\nEvaluating competitive agents...")
evaluate_agents(coop=False)
```

PROF

Key Concepts in This Implementation:

1. **Multi-Agent Environment**: The custom environment supports both cooperative and competitive scenarios through a config flag.
2. **Policy Sharing**: Agents can share policies or have individual policies based on the configuration.
3. **Custom Neural Network**: A PyTorch model is defined for policy approximation.
4. **Training Configuration**: RLlib's PPO algorithm is used with appropriate multi-agent settings.
5. **Evaluation**: Separate evaluation function to test trained agents.

This setup can be extended with:

- More complex environment dynamics
- Additional agents
- Different reward structures
- Hierarchical or communication-based policies
- Parameter sharing between agents

Remember to adjust hyperparameters (learning rate, batch sizes, etc.) based on your specific requirements and environment complexity.

Solving CartPole with Deep Q-Network (DQN) in PyTorch

Below is a complete implementation of a DQN agent to solve the CartPole-v1 environment from OpenAI Gym.

1. Import Required Libraries


```
import gym
import numpy as np
import random
from collections import deque
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import matplotlib.pyplot as plt
```

2. Define the Q-Network

```
class DQN(nn.Module):
    def __init__(self, state_size, action_size):
        super(DQN, self).__init__()
        self.fc1 = nn.Linear(state_size, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, action_size)

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

3. Define the DQN Agent

```
class DQNAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.memory = deque(maxlen=10000)
        self.gamma = 0.95    # discount rate
        self.epsilon = 1.0   # exploration rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = 0.001
        self.model = DQN(state_size, action_size)
        self.optimizer = optim.Adam(self.model.parameters(),
lr=self.learning_rate)

    def remember(self, state, action, reward, next_state, done):
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state):
        if np.random.rand() <= self.epsilon:
            return random.randrange(self.action_size)
        state = torch.FloatTensor(state)
        act_values = self.model(state)
        return torch.argmax(act_values).item()

    def replay(self, batch_size):
        if len(self.memory) < batch_size:
            return

        minibatch = random.sample(self.memory, batch_size)

        states = torch.FloatTensor(np.array([t[0] for t in minibatch]))
        actions = torch.LongTensor(np.array([t[1] for t in minibatch]))
        rewards = torch.FloatTensor(np.array([t[2] for t in minibatch]))
        next_states = torch.FloatTensor(np.array([t[3] for t in
minibatch]))
```

```
        dones = torch.FloatTensor(np.array([t[4] for t in minibatch]))

        # Current Q values
        current_q = self.model(states).gather(1, actions.unsqueeze(1))

        # Next Q values
        next_q = self.model(next_states).max(1)[0].detach()
        target_q = rewards + (1 - dones) * self.gamma * next_q

        # Compute loss and update
        loss = F.mse_loss(current_q.squeeze(), target_q)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # Decay epsilon
        if self.epsilon > self.epsilon_min:
            self.epsilon *= self.epsilon_decay

    def save(self, filename):
        torch.save(self.model.state_dict(), filename)

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

4. Training Loop

```
def train_dqn(episodes=1000, batch_size=32):
    env = gym.make('CartPole-v1')
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n

    agent = DQNAgent(state_size, action_size)
    scores = []

    for e in range(episodes):
        state = env.reset()
        state = np.array(state)
        total_reward = 0

        for time in range(500):  # Max steps per episode
            # env.render()  # Uncomment to visualize training
            action = agent.act(state)
            next_state, reward, done, _ = env.step(action)
            next_state = np.array(next_state)

            # Custom reward shaping can be added here if needed
            reward = reward if not done else -10

            agent.remember(state, action, reward, next_state, done)
```

PROF

```
            state = next_state
            total_reward += reward

            if done:
                break

        scores.append(total_reward)
        agent.replay(batch_size)

        # Print progress
        avg_score = np.mean(scores[-100:])  # Last 100 episodes
        print(f"Episode: {e+1}/{episodes}, Score: {total_reward}, Avg Score: {avg_score:.2f}, Epsilon: {agent.epsilon:.2f}")

        # Early stopping if solved
        if avg_score >= 195:  # CartPole-v1 solved condition
            print(f"Solved in {e+1} episodes!")
            agent.save('cartpole_dqn.pth')
            break

    env.close()
    return scores
```

5. Run Training and Plot Results

```
if __name__ == "__main__":
    # Train the agent
    scores = train_dqn(episodes=500)

    # Plot the learning progress
    plt.figure(figsize=(10, 5))
    plt.plot(scores)
    plt.title('DQN Training Progress')
    plt.xlabel('Episode')
    plt.ylabel('Score')
    plt.grid(True)

    # Add moving average
    window_size = 100
    moving_avg = np.convolve(scores, np.ones(window_size)/window_size, mode='valid')
    plt.plot(moving_avg, color='red', label=f'{window_size}-episode average')
    plt.legend()

    plt.show()
```

PROF

6. Evaluate the Trained Agent

```
def evaluate_agent():
    env = gym.make('CartPole-v1')
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n

    agent = DQNAgent(state_size, action_size)
    agent.load('cartpole_dqn.pth')
    agent.epsilon = 0  # No exploration during evaluation

    total_rewards = []

    for e in range(10):  # Run 10 evaluation episodes
        state = env.reset()
        state = np.array(state)
        total_reward = 0

        for time in range(500):
            env.render()  # Visualize the trained agent
            action = agent.act(state)
            next_state, reward, done, _ = env.step(action)
            state = np.array(next_state)
            total_reward += reward

            if done:
                break

        total_rewards.append(total_reward)
        print(f"Evaluation Episode {e+1}, Score: {total_reward}")

    env.close()
    print(f"\nAverage score over 10 episodes:
{np.mean(total_rewards):.2f}")

# Uncomment to evaluate
# evaluate_agent()
```


Key Components Explained:

1. **Q-Network**: A simple 3-layer neural network that approximates the Q-function.
2. **Experience Replay**: Stores transitions (state, action, reward, next_state) in a replay buffer for more efficient learning.
3. **Epsilon-Greedy Policy**: Balances exploration (random actions) and exploitation (actions with highest Q-values).
4. **Target Network**: While not implemented here (for simplicity), a common DQN improvement is to use a separate target network that's updated less frequently.

5. **Training Loop**: The agent interacts with the environment, stores experiences, and learns from random batches of these experiences.

Potential Improvements:

1. **Target Network**: Add a separate target network that's updated periodically to stabilize training.
2. **Double DQN**: Decouple action selection from value estimation to reduce overestimation bias.
3. **Prioritized Experience Replay**: Sample important transitions more frequently.
4. **Dueling DQN**: Separate the network into value and advantage streams.
5. **Hyperparameter Tuning**: Adjust learning rate, batch size, network architecture, etc.

This implementation should solve CartPole-v1 (reach an average score of 195+ over 100 episodes) within 200-300 episodes. The evaluation script lets you watch your trained agent balance the pole.

Training a PPO Agent on LunarLander-v2 with PyTorch

Here's a complete implementation of Proximal Policy Optimization (PPO) for the LunarLander-v2 environment:

1. Import Required Libraries

```
import gym
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.distributions import Categorical
import matplotlib.pyplot as plt
from collections import deque
```


2. Define the Policy Network

```
class PolicyNetwork(nn.Module):
    def __init__(self, state_size, action_size, hidden_size=64):
        super(PolicyNetwork, self).__init__()
        self.fc1 = nn.Linear(state_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, action_size)

        # Value function estimator
        self.fc_v1 = nn.Linear(state_size, hidden_size)
```

```
        self.fc_v2 = nn.Linear(hidden_size, hidden_size)
        self.fc_v3 = nn.Linear(hidden_size, 1)

    def forward(self, x):
        # Actor network
        x_actor = F.relu(self.fc1(x))
        x_actor = F.relu(self.fc2(x_actor))
        logits = self.fc3(x_actor)

        # Critic network
        x_critic = F.relu(self.fc_v1(x))
        x_critic = F.relu(self.fc_v2(x_critic))
        value = self.fc_v3(x_critic)

        return logits, value
```

3. Define the PPO Agent

```
class PPOAgent:
    def __init__(self, state_size, action_size):
        self.state_size = state_size
        self.action_size = action_size
        self.gamma = 0.99        # discount factor
        self.epsilon = 0.2       # clipping parameter
        self.lr = 3e-4           # learning rate
        self.beta = 0.01         # entropy coefficient
        self.update_epochs = 10  # number of epochs per update
        self.clip_grad = 0.5     # gradient clipping

        self.policy = PolicyNetwork(state_size, action_size)
        self.optimizer = optim.Adam(self.policy.parameters(),
lr=self.lr)
        self.memory = deque()

    def act(self, state):
        state = torch.FloatTensor(state)
        logits, value = self.policy(state)
        dist = Categorical(logits=logits)
        action = dist.sample()
        log_prob = dist.log_prob(action)

        return action.item(), log_prob.item(), value.item()

    def remember(self, state, action, log_prob, value, reward, done):
        self.memory.append((state, action, log_prob, value, reward,
done))

    def compute_returns(self, rewards, dones, last_value):
        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):
        if len(self.memory) < 1:
            return

        # Unzip the memory
        states, actions, old_log_probs, values, rewards, dones =
zip(*self.memory)

        # Convert to tensors
        states = torch.FloatTensor(np.array(states))
        actions = torch.LongTensor(np.array(actions))
        old_log_probs = torch.FloatTensor(np.array(old_log_probs))
        old_values = torch.FloatTensor(np.array(values))
        rewards = torch.FloatTensor(np.array(rewards))
        dones = torch.FloatTensor(np.array(dones))

        # Compute returns and advantages
        returns = torch.FloatTensor(self.compute_returns(rewards, dones,
old_values[-1]))
        advantages = returns - old_values

        # Normalize advantages
        advantages = (advantages - advantages.mean()) /
(advantages.std() + 1e-8)

        # Optimize policy for K epochs
        for _ in range(self.update_epochs):
            # Get new log probs and values
            logits, new_values = self.policy(states)
            dist = Categorical(logits=logits)
            new_log_probs = dist.log_prob(actions)

            # Calculate ratio (pi_theta / pi_theta_old)
            ratio = (new_log_probs - old_log_probs).exp()

            # Surrogate losses
            surr1 = ratio * advantages
            surr2 = torch.clamp(ratio, 1.0 - self.epsilon, 1.0 +
self.epsilon) * advantages

            # Actor loss
            actor_loss = -torch.min(surr1, surr2).mean()

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

            # Entropy bonus
            entropy = dist.entropy().mean()
```

PROF

```
            # Total loss
            loss = actor_loss + 0.5 * critic_loss - self.beta * entropy

            # 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()

    def save(self, filename):
        torch.save(self.policy.state_dict(), filename)

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

4. Training Loop

```
def train_ppo(episodes=1000, max_steps=1000, batch_size=2048):
    env = gym.make('LunarLander-v2')
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n

    agent = PPOAgent(state_size, action_size)
    scores = []
    avg_scores = []

    for e in range(1, episodes+1):
        state = env.reset()
        score = 0
        steps = 0

        while True:
            # Collect experience
            action, log_prob, value = agent.act(state)
            next_state, reward, done, _ = env.step(action)

            agent.remember(state, action, log_prob, value, reward, done)
            state = next_state
            score += reward
            steps += 1

            # Update if we've collected enough samples
            if steps % batch_size == 0 or done:
                agent.update()
```

```
            if done:
                break

        scores.append(score)
        avg_score = np.mean(scores[-100:])
        avg_scores.append(avg_score)

        print(f"Episode: {e}, Score: {score:.2f}, Avg Score: {avg_score:.2f}")

        # Early stopping if solved (avg score > 200 over last 100 episodes)
        if avg_score >= 200:
            print(f"Solved in {e} episodes!")
            agent.save('lunarlander_ppo.pth')
            break

    env.close()
    return scores, avg_scores
```

5. Run Training and Plot Results

```
if __name__ == "__main__":
    # Train the agent
    scores, avg_scores = train_ppo(episodes=1000)

    # Plot the results
    plt.figure(figsize=(12, 6))
    plt.plot(scores, alpha=0.5, label='Episode Score')
    plt.plot(avg_scores, label='100-Episode Average')
    plt.axhline(y=200, color='r', linestyle='-', label='Solved Threshold')
    plt.title('PPO Training Progress on LunarLander-v2')
    plt.xlabel('Episode')
    plt.ylabel('Score')
    plt.legend()
    plt.grid(True)
    plt.show()
```


6. Evaluate the Trained Agent

```
def evaluate_ppo(episodes=10):
    env = gym.make('LunarLander-v2')
    state_size = env.observation_space.shape[0]
    action_size = env.action_space.n

    agent = PPOAgent(state_size, action_size)
    agent.load('lunarlander_ppo.pth')
```

```
    scores = []

    for e in range(1, episodes+1):
        state = env.reset()
        score = 0
        done = False

        while not done:
            env.render()
            action, _, _ = agent.act(state)
            state, reward, done, _ = env.step(action)
            score += reward

        scores.append(score)
        print(f"Evaluation Episode {e}, Score: {score:.2f}")

    env.close()
    print(f"\nAverage score over {episodes} episodes:
{np.mean(scores):.2f}")

# Uncomment to evaluate
# evaluate_ppo()
```

Key Components Explained:

1. **Policy Network**: Contains both actor (policy) and critic (value function) heads.
2. **Experience Collection**: Stores transitions (state, action, log_prob, value, reward, done) in memory.
3. **Generalized Advantage Estimation**: Computes advantages using the critic's value estimates.
4. **Clipped Objective**: The PPO loss function includes clipping to prevent large policy updates.
5. **Multiple Optimization Epochs**: Performs several epochs of optimization on the same batch of data.

Hyperparameter Tuning Tips:

1. **Learning Rate**: Start with 3e-4 (common for Adam optimizer) and adjust if needed.
2. **Batch Size**: Larger batches (2048-4096) often work well for PPO.
3. **Gamma**: 0.99 is standard, but can be adjusted for longer/shorter horizons.
4. **Clipping Parameter (ε)**: Typically between 0.1-0.3.
5. **Entropy Coefficient**: Helps with exploration (start with 0.01).

Potential Improvements:

1. **Parallel Environments**: Collect experience from multiple environments simultaneously.

2. **Normalization**: Add observation normalization and advantage normalization.
3. **Learning Rate Scheduling**: Gradually decrease learning rate during training.
4. **Network Architecture**: Experiment with different network sizes and architectures.
5. **Hyperparameter Optimization**: Use tools like Optuna for automated tuning.

This implementation should solve LunarLander-v2 (reach an average score of 200+ over 100 episodes) within 300-500 episodes. The evaluation script lets you watch your trained agent land the lunar module.

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
        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()

        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)
            # 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
```

PROF

```
        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':
```

PROF

```
            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
```

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)
```

PROF

```
            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"""
        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)
```

PROF

```
            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
```

PROF

```
            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()

# 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

Monte Carlo Tree Search (MCTS) with Neural Network in PyTorch

Below is a complete implementation combining MCTS with a neural network (often called "AlphaZero-style" architecture) for game playing, using PyTorch.

1. Core Components

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from collections import defaultdict, namedtuple
import math
import random

# Data structure for MCTS nodes
MCTSNode = namedtuple('MCTSNode', ['state', 'parent', 'action',
'children', 'visit_count', 'total_value', 'prior'])
```

2. Neural Network Architecture

```
class AlphaZeroNet(nn.Module):
    """Neural network that combines policy and value estimation"""

    def __init__(self, game, hidden_size=256):
        super().__init__()
        self.game = game

        # Input block
        self.conv1 = nn.Conv2d(game.input_channels, hidden_size//4,
kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(hidden_size//4)

        # Residual blocks
        self.residual_blocks = nn.ModuleList([
            nn.Sequential(
                nn.Conv2d(hidden_size//4, hidden_size//4, kernel_size=3,
padding=1),
                nn.BatchNorm2d(hidden_size//4),
                nn.ReLU(),
                nn.Conv2d(hidden_size//4, hidden_size//4, kernel_size=3,
```

```
padding=1),
                nn.BatchNorm2d(hidden_size//4)
            ) for _ in range(5)
        ])

        # Policy head
        self.policy_conv = nn.Conv2d(hidden_size//4, 2, kernel_size=1)
        self.policy_bn = nn.BatchNorm2d(2)
        self.policy_fc = nn.Linear(2 * game.board_size**2,
game.action_size)

        # Value head
        self.value_conv = nn.Conv2d(hidden_size//4, 1, kernel_size=1)
        self.value_bn = nn.BatchNorm2d(1)
        self.value_fc1 = nn.Linear(game.board_size**2, hidden_size//2)
        self.value_fc2 = nn.Linear(hidden_size//2, 1)

    def forward(self, x):
        # Input block
        x = F.relu(self.bn1(self.conv1(x)))

        # Residual blocks
        for block in self.residual_blocks:
            residual = x
            x = F.relu(block(x))
            x += residual
            x = F.relu(x)

        # Policy head
        policy = F.relu(self.policy_bn(self.policy_conv(x)))
        policy = policy.view(policy.size(0), -1)
        policy = self.policy_fc(policy)
        policy = F.softmax(policy, dim=1)

        # Value head
        value = F.relu(self.value_bn(self.value_conv(x)))
        value = value.view(value.size(0), -1)
        value = F.relu(self.value_fc1(value))
        value = torch.tanh(self.value_fc2(value))

        return policy, value

    def predict(self, state):
        """Convenience method for predictions"""
        with torch.no_grad():
            state_tensor = self.game.state_to_tensor(state)
            policy, value = self.forward(state_tensor.unsqueeze(0))
        return policy.squeeze(0).cpu().numpy(), value.item()
```

PROF

3. Monte Carlo Tree Search Implementation

```python
class MCTS:
    """Monte Carlo Tree Search with neural network guidance"""

    def __init__(self, game, model, args):
        self.game = game
        self.model = model
        self.args = args
        self.Q = defaultdict(float)  # Total action value
        self.N = defaultdict(int)    # Visit count
        self.P = defaultdict(float)  # Prior probabilities

    def search(self, state, num_simulations=800):
        """Perform MCTS simulations from given state"""

        root = MCTSNode(
            state=state,
            parent=None,
            action=None,
            children=[],
            visit_count=0,
            total_value=0,
            prior=0
        )

        for _ in range(num_simulations):
            node = root
            search_path = [node]

            # Selection
            while node.children:
                node = self.select_child(node)
                search_path.append(node)

            # Expansion
            if not self.game.is_terminal(node.state):
                policy, value = self.model.predict(node.state)
                valid_actions = self.game.get_valid_actions(node.state)
                policy = policy * valid_actions
                policy /= np.sum(policy)

                for action in range(self.game.action_size):
                    if valid_actions[action]:
                        child_state =
self.game.get_next_state(node.state, action)
                        child_node = MCTSNode(
                            state=child_state,
                            parent=node,
                            action=action,
                            children=[],
                            visit_count=0,
                            total_value=0,
                            prior=policy[action]
```

PROF

```
                        )
                        node.children.append(child_node)

            # Backpropagation
            value = self.evaluate(node)
            self.backpropagate(search_path, value)

        return root

    def select_child(self, node):
        """Select child node using UCB formula"""
        total_visits = sum(child.visit_count for child in node.children)
        log_total_visits = math.log(total_visits + 1e-10)

        def ucb_score(child):
            q = child.total_value / (child.visit_count + 1e-10)
            u = self.args.c_puct * child.prior *
math.sqrt(log_total_visits) / (child.visit_count + 1)
            return q + u

        return max(node.children, key=ucb_score)

    def evaluate(self, node):
        """Evaluate a leaf node"""
        if self.game.is_terminal(node.state):
            return self.game.get_reward(node.state)
        else:
            _, value = self.model.predict(node.state)
            return value

    def backpropagate(self, path, value):
        """Backpropagate the value through the search path"""
        for node in reversed(path):
            node.visit_count += 1
            node.total_value += value
            value = -value  # Alternate values for alternating players

    def get_action_probs(self, root, temperature=1):
        """Get action probabilities from root node visit counts"""
        visit_counts = np.array([child.visit_count for child in
root.children])
        if temperature == 0:
            probs = np.zeros_like(visit_counts)
            probs[np.argmax(visit_counts)] = 1
        else:
            probs = visit_counts ** (1 / temperature)
            probs /= probs.sum()
        return probs
```


4. Game Interface (Abstract Class)

```python
class Game:
    """Abstract class defining the game interface"""

    @property
    def board_size(self):
        """Size of the game board (n x n)"""
        raise NotImplementedError

    @property
    def action_size(self):
        """Number of possible actions"""
        raise NotImplementedError

    @property
    def input_channels(self):
        """Number of input channels for the neural network"""
        raise NotImplementedError

    def get_init_state(self):
        """Get initial game state"""
        raise NotImplementedError

    def get_next_state(self, state, action):
        """Get next state after taking action"""
        raise NotImplementedError

    def get_valid_actions(self, state):
        """Get mask of valid actions (1=valid, 0=invalid)"""
        raise NotImplementedError

    def is_terminal(self, state):
        """Check if state is terminal"""
        raise NotImplementedError

    def get_reward(self, state):
        """Get reward for terminal state (from current player's perspective)"""
        raise NotImplementedError

    def state_to_tensor(self, state):
        """Convert state to neural network input tensor"""
        raise NotImplementedError

    def display(self, state):
        """Display the current state"""
        raise NotImplementedError
```


5. Training Loop

```
class Trainer:
    """Class to train the neural network with MCTS"""

    def __init__(self, game, model, args):
        self.game = game
        self.model = model
        self.args = args
        self.mcts = MCTS(game, model, args)
        self.optimizer = optim.Adam(model.parameters(),
lr=args.learning_rate)
        self.memory = deque(maxlen=args.memory_size)

    def self_play(self):
        """Generate training data through self-play"""
        state = self.game.get_init_state()
        states = []
        probs = []
        rewards = []

        while True:
            # Run MCTS from current state
            root = self.mcts.search(state, self.args.num_simulations)

            # Get action probabilities
            action_probs = self.mcts.get_action_probs(root,
self.args.temperature)

            # Store training data
            states.append(state)
            probs.append(action_probs)

            # Choose action
            action = np.random.choice(len(action_probs), p=action_probs)

            # Take action
            state = self.game.get_next_state(state, action)

            if self.game.is_terminal(state):
                # Determine final rewards
                reward = self.game.get_reward(state)
                rewards = [reward * ((-1) ** i) for i in
range(len(states))]
                break

        # Prepare training examples
        examples = []
        for state, prob, reward in zip(states, probs, rewards):
            examples.append((state, prob, reward))

        return examples

    def train(self):
```

PROF

```
        """Train the neural network"""
        if len(self.memory) < self.args.batch_size:
            return

        # Sample batch
        batch = random.sample(self.memory, self.args.batch_size)
        states, target_probs, target_values = zip(*batch)

        # Convert to tensors
        state_tensors = torch.stack([self.game.state_to_tensor(s) for s
in states])
        target_probs = torch.FloatTensor(np.array(target_probs))
        target_values =
torch.FloatTensor(np.array(target_values)).unsqueeze(1)

        # Forward pass
        policy_logits, values = self.model(state_tensors)

        # Compute losses
        policy_loss = -torch.mean(torch.sum(target_probs *
torch.log_softmax(policy_logits, dim=1), dim=1))
        value_loss = F.mse_loss(values, target_values)
        loss = policy_loss + value_loss

        # Backward pass
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        return {
            'total_loss': loss.item(),
            'policy_loss': policy_loss.item(),
            'value_loss': value_loss.item()
        }

    def learn(self):
        """Main training loop"""
        for iteration in range(1, self.args.num_iterations + 1):
            # Self-play to generate data
            examples = self.self_play()
            self.memory.extend(examples)

            # Train network
            loss_info = self.train()

            # Evaluation
            if iteration % self.args.eval_interval == 0:
                win_rate = self.evaluate()
                print(f"Iteration {iteration}: Loss=
{loss_info['total_loss']:.4f}, Win Rate={win_rate:.2f}")

                # Save model
                torch.save(self.model.state_dict(),
```

```
f"az_model_{iteration}.pth")

    def evaluate(self, num_games=20):
        """Evaluate current model against random player"""
        wins = 0

        for _ in range(num_games):
            state = self.game.get_init_state()
            current_player = 1

            while True:
                if current_player == 1:
                    # Use MCTS for our player
                    root = self.mcts.search(state,
self.args.num_simulations//2)
                    action = max(root.children, key=lambda c:
c.visit_count).action
                else:
                    # Random player
                    valid_actions = self.game.get_valid_actions(state)
                    action = np.random.choice(np.where(valid_actions)
[0])

                state = self.game.get_next_state(state, action)

                if self.game.is_terminal(state):
                    reward = self.game.get_reward(state)
                    if reward > 0:
                        wins += 1
                    break

                current_player *= -1

        return wins / num_games
```

PROF

6. Example Configuration

```
class Config:
    """Configuration for training"""
    def __init__(self):
        self.c_puct = 1.0                   # Exploration constant
        self.num_simulations = 800           # MCTS simulations per move
        self.num_iterations = 1000           # Training iterations
        self.memory_size = 100000            # Replay buffer size
        self.batch_size = 1024               # Training batch size
        self.learning_rate = 0.001           # Learning rate
        self.temperature = 1.0               # Temperature for action
selection
        self.eval_interval = 10              # Evaluation interval
```

7. Example Usage with Tic-Tac-Toe

```
class TicTacToe(Game):
    """Implementation of Tic-Tac-Toe for the AlphaZero framework"""

    def __init__(self):
        self.size = 3
        self.action_size = self.size ** 2
        self.input_channels = 3  # Current player, player 1, player 2

    @property
    def board_size(self):
        return self.size

    def get_init_state(self):
        return np.zeros((self.size, self.size), dtype=int)

    def get_next_state(self, state, action):
        next_state = state.copy()
        row, col = action // self.size, action % self.size
        player = self.get_current_player(state)
        next_state[row, col] = player
        return next_state

    def get_valid_actions(self, state):
        return (state.reshape(-1) == 0).astype(int)

    def is_terminal(self, state):
        # Check rows
        for row in range(self.size):
            if abs(sum(state[row, :])) == self.size and state[row, 0] != 0:
                return True

        # Check columns
        for col in range(self.size):
            if abs(sum(state[:, col])) == self.size and state[0, col] != 0:
                return True

        # Check diagonals
        if abs(sum(state[i, i] for i in range(self.size))) == self.size and state[0, 0] != 0:
            return True
        if abs(sum(state[i, self.size-1-i] for i in range(self.size))) == self.size and state[0, self.size-1] != 0:
            return True

        # Check if board is full
        return np.all(state != 0)
```

```
    def get_reward(self, state):
        # Check rows
        for row in range(self.size):
            if abs(sum(state[row, :])) == self.size:
                return np.sign(state[row, 0])

        # Check columns
        for col in range(self.size):
            if abs(sum(state[:, col])) == self.size:
                return np.sign(state[0, col])

        # Check diagonals
        if abs(sum(state[i, i] for i in range(self.size))) == self.size:
            return np.sign(state[0, 0])
        if abs(sum(state[i, self.size-1-i] for i in range(self.size))) == self.size:
            return np.sign(state[0, self.size-1])

        # Draw
        return 0

    def get_current_player(self, state):
        return 1 if np.sum(state != 0) % 2 == 0 else -1

    def state_to_tensor(self, state):
        current_player = self.get_current_player(state)
        tensor = np.zeros((self.input_channels, self.size, self.size), dtype=np.float32)
        tensor[0] = current_player
        tensor[1] = (state == 1).astype(np.float32)
        tensor[2] = (state == -1).astype(np.float32)
        return torch.from_numpy(tensor)

    def display(self, state):
        symbols = {0: '.', 1: 'X', -1: 'O'}
        for row in range(self.size):
            print(' '.join(symbols[state[row, col]] for col in range(self.size)))
        print()

# Example training
if __name__ == "__main__":
    game = TicTacToe()
    model = AlphaZeroNet(game)
    args = Config()

    trainer = Trainer(game, model, args)
    trainer.learn()
```

PROF

Key Features:

1. **Neural Network Integration**:
   - Dual-output network (policy + value)
   - Residual network architecture
   - Efficient state representation
2. **MCTS Implementation**:
   - UCB1 selection with neural network priors
   - Parallelizable simulations
   - Temperature-controlled action selection
3. **Training Pipeline**:
   - Self-play data generation
   - Experience replay buffer
   - Periodic evaluation
4. **Modular Design**:
   - Abstract Game interface for easy adaptation
   - Configurable hyperparameters
   - Separate components for easy modification

This implementation can be adapted to various games by implementing the Game interface. The neural network will learn to evaluate positions and suggest moves, while MCTS provides the search capability to explore possible future states.

Imitation Learning (Behavioral Cloning) in PyTorch

Below is a complete implementation of Behavioral Cloning, a form of imitation learning, using PyTorch. This approach learns a policy by cloning expert demonstrations.


```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import numpy as np
import os
from collections import deque
import random

# 1. Define the Neural Network Policy
class PolicyNetwork(nn.Module):
    def __init__(self, state_dim, action_dim, hidden_size=256):
        super(PolicyNetwork, self).__init__()
        self.fc1 = nn.Linear(state_dim, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
```

```python
        self.fc3 = nn.Linear(hidden_size, action_dim)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()  # For bounded action spaces

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.tanh(self.fc3(x))  # Assuming actions are in [-1, 1]
        return x

# 2. Create Dataset for Expert Demonstrations
class ExpertDataset(Dataset):
    def __init__(self, states, actions):
        self.states = states
        self.actions = actions

    def __len__(self):
        return len(self.states)

    def __getitem__(self, idx):
        return self.states[idx], self.actions[idx]

# 3. Behavioral Cloning Agent
class BCAgent:
    def __init__(self, state_dim, action_dim, lr=1e-3, batch_size=64):
        self.policy = PolicyNetwork(state_dim, action_dim)
        self.optimizer = optim.Adam(self.policy.parameters(), lr=lr)
        self.criterion = nn.MSELoss()  # For continuous actions
        self.batch_size = batch_size
        self.device = torch.device("cuda" if torch.cuda.is_available()
else "cpu")
        self.policy.to(self.device)

    def train(self, expert_states, expert_actions, epochs=100):
        # Create dataset and dataloader
        dataset = ExpertDataset(expert_states, expert_actions)
        dataloader = DataLoader(dataset, batch_size=self.batch_size,
shuffle=True)

        losses = []
        for epoch in range(epochs):
            epoch_loss = 0
            for batch_states, batch_actions in dataloader:
                # Move data to device
                batch_states = batch_states.float().to(self.device)
                batch_actions = batch_actions.float().to(self.device)

                # Forward pass
                pred_actions = self.policy(batch_states)

                # Compute loss
                loss = self.criterion(pred_actions, batch_actions)
```

```
                # Backward pass
                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()

                epoch_loss += loss.item()

            avg_loss = epoch_loss / len(dataloader)
            losses.append(avg_loss)
            if epoch % 10 == 0:
                print(f"Epoch {epoch}, Loss: {avg_loss:.4f}")

        return losses

    def act(self, state):
        state = torch.FloatTensor(state).unsqueeze(0).to(self.device)
        with torch.no_grad():
            action = self.policy(state)
        return action.cpu().numpy()[0]

    def save(self, path):
        torch.save(self.policy.state_dict(), path)

    def load(self, path):
        self.policy.load_state_dict(torch.load(path))
        self.policy.eval()

# 4. Example Usage
if __name__ == "__main__":
    # Hyperparameters
    STATE_DIM = 8  # Example: LunarLander-v2 has 8 state dimensions
    ACTION_DIM = 2  # LunarLander-v2 has 2 continuous actions
    LR = 3e-4
    BATCH_SIZE = 128
    EPOCHS = 100

    # Create agent
    agent = BCAgent(STATE_DIM, ACTION_DIM, lr=LR, batch_size=BATCH_SIZE)

    # Generate or load expert demonstrations
    # In practice, you would collect these from an expert policy
    NUM_DEMOS = 10000
    expert_states = np.random.randn(NUM_DEMOS, STATE_DIM)  # Random
states for example
    expert_actions = np.random.randn(NUM_DEMOS, ACTION_DIM) * 0.1  #
Small random actions

    # Train the agent
    losses = agent.train(expert_states, expert_actions, epochs=EPOCHS)

    # Test the trained agent
    test_state = np.random.randn(STATE_DIM)
    action = agent.act(test_state)
```

PROF

```
    print(f"Test action: {action}")

    # Save the trained model
    agent.save("bc_policy.pth")
```

Key Components Explained:

1. **Policy Network**: A simple feedforward neural network that maps states to actions.

2. **Expert Dataset**: Stores the state-action pairs from expert demonstrations.

3. **BCAgent Class**:

    - `train()`: Trains the policy network to mimic expert actions using supervised learning
    - `act()`: Uses the trained policy to select actions given states
    - Save/load functionality for the policy

4. **Training Process**:

    - The agent learns by minimizing the MSE between its predicted actions and the expert actions
    - Standard supervised learning approach with batches

For Real-World Usage:

1. Replace the random expert demonstrations with real expert data from your domain
2. Adjust the network architecture based on your problem complexity
3. For discrete actions, modify the output layer to use softmax and cross-entropy loss
4. Consider adding data augmentation or regularization techniques

Extensions:

1. **Dataset Aggregation (DAgger)**: Iteratively collect more data by having the trained policy interact with the environment and get corrected by the expert
2. **Ensemble Methods**: Train multiple policies to reduce compounding errors
3. **Additional Losses**: Incorporate environment-specific losses or regularization

---

Integrating LLMs with RL for Decision-Making in PyTorch

Below is a comprehensive implementation that combines Large Language Models (LLMs) with Reinforcement Learning (RL) for decision-making tasks. This approach uses the LLM to guide or augment the RL agent's policy.

Approach Overview

1. **LLM-as-Policy**: Use the LLM directly as the policy network
2. **LLM-as-Reward**: Use the LLM to provide additional reward signals
3. **LLM-as-Advisor**: Use the LLM to suggest actions during training

```
import torch
import torch.nn as nn
import torch.optim as optim
from transformers import AutoModel, AutoTokenizer
from collections import deque
import numpy as np
import random

# 1. LLM-Augmented Policy Network
class LLMPolicyNetwork(nn.Module):
    def __init__(self, state_dim, action_dim, llm_model_name="bert-base-uncased"):
        super(LLMPolicyNetwork, self).__init__()

        # Pretrained LLM
        self.llm = AutoModel.from_pretrained(llm_model_name)
        self.llm_tokenizer = AutoTokenizer.from_pretrained(llm_model_name)

        # Freeze LLM parameters (optional)
        for param in self.llm.parameters():
            param.requires_grad = False

        # RL policy head
        self.state_proj = nn.Linear(state_dim, self.llm.config.hidden_size)
        self.policy_head = nn.Linear(self.llm.config.hidden_size, action_dim)
        self.tanh = nn.Tanh()

    def forward(self, state, text_input=None):
        # Process state
        state_emb = self.state_proj(state)

        # Process text input if provided
        if text_input is not None:
            text_encoding = self.llm_tokenizer(text_input, return_tensors='pt', padding=True, truncation=True)
            text_emb = self.llm(**text_encoding).last_hidden_state.mean(dim=1)
            combined_emb = state_emb + text_emb
        else:
            combined_emb = state_emb

        # Get action
        action = self.tanh(self.policy_head(combined_emb))
        return action
```

```
# 2. LLM-Augmented RL Agent
class LLMRLAgent:
    def __init__(self, state_dim, action_dim, lr=1e-4, gamma=0.99):
        self.device = torch.device("cuda" if torch.cuda.is_available()
else "cpu")

        # Policy network
        self.policy = LLMPolicyNetwork(state_dim,
action_dim).to(self.device)
        self.optimizer = optim.Adam(self.policy.parameters(), lr=lr)

        # RL parameters
        self.gamma = gamma
        self.memory = deque(maxlen=10000)

        # Action space
        self.action_dim = action_dim

    def act(self, state, text_input=None, epsilon=0.1):
        state = torch.FloatTensor(state).unsqueeze(0).to(self.device)

        # Epsilon-greedy exploration
        if random.random() < epsilon:
            return np.random.uniform(-1, 1, self.action_dim)

        with torch.no_grad():
            action = self.policy(state, text_input)
        return action.cpu().numpy()[0]

    def remember(self, state, action, reward, next_state, done,
text_input=None):
        self.memory.append((state, action, reward, next_state, done,
text_input))

    def _get_llm_reward(self, state, action, next_state):
        """Use LLM to generate additional reward signal"""
        # In practice, you would implement a meaningful reward function
        # based on your task and LLM capabilities
        return 0.0

    def train(self, batch_size=64):
        if len(self.memory) < batch_size:
            return

        # Sample batch
        batch = random.sample(self.memory, batch_size)
        states, actions, rewards, next_states, dones, text_inputs =
zip(*batch)

        # Convert to tensors
        states = torch.FloatTensor(np.array(states)).to(self.device)
        actions = torch.FloatTensor(np.array(actions)).to(self.device)
```

PROF

```
        rewards = torch.FloatTensor(np.array(rewards)).to(self.device)
        next_states =
torch.FloatTensor(np.array(next_states)).to(self.device)
        dones = torch.FloatTensor(np.array(dones)).to(self.device)

        # Current Q values
        current_q = self.policy(states, text_inputs)

        # Target Q values
        with torch.no_grad():
            next_q = self.policy(next_states, text_inputs)
            target_q = rewards + (1 - dones) * self.gamma * next_q

        # Compute loss
        loss = nn.MSELoss()(current_q, target_q)

        # Optimize
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        return loss.item()

    def save(self, path):
        torch.save(self.policy.state_dict(), path)

    def load(self, path):
        self.policy.load_state_dict(torch.load(path))
        self.policy.eval()

# 3. Example Usage
if __name__ == "__main__":
    # Environment parameters
    STATE_DIM = 10
    ACTION_DIM = 2

    # Create agent
    agent = LLMRLAgent(STATE_DIM, ACTION_DIM)

    # Training loop example
    for episode in range(100):
        state = np.random.randn(STATE_DIM)  # Simulated environment
        text_input = "Move toward the target while avoiding obstacles"
# LLM guidance

        total_reward = 0
        done = False

        while not done:
            # Get action
            action = agent.act(state, text_input)

            # Simulate environment step
```

```
            next_state = state + np.random.randn(STATE_DIM) * 0.1
            reward = -np.linalg.norm(next_state)  # Simple reward
            done = np.random.rand() < 0.05  # 5% chance of episode ending

            # Add LLM reward
            llm_reward = agent._get_llm_reward(state, action, next_state)
            total_reward += reward + llm_reward

            # Remember experience
            agent.remember(state, action, reward + llm_reward, next_state, done, text_input)

            # Train
            loss = agent.train()

            state = next_state

        print(f"Episode {episode}, Total Reward: {total_reward:.2f}, Loss: {loss:.4f}")

    # Save trained model
    agent.save("llm_rl_agent.pth")
```

Key Integration Strategies

1. LLM as Policy Component

The `LLMPolicyNetwork` combines:

- Traditional state inputs (processed through dense layers)
- Text inputs (processed through the LLM)
- Combined representation used for action selection

2. LLM for Reward Shaping

The `_get_llm_reward` method demonstrates how to use an LLM to:

- Provide additional reward signals based on semantic understanding
- Align agent behavior with natural language instructions
- Incorporate human preferences or safety constraints

3. LLM for Action Advising

The `act` method shows how to:

- Use natural language prompts to guide the agent
- Combine learned RL policies with LLM suggestions
- Implement exploration strategies that consider LLM outputs

Advanced Integration Options

1. **Prompt Engineering**:

```
def generate_llm_prompt(state):
    """Convert state to natural language prompt"""
    return f"The agent is at position {state[:2]}. There are objects at
{state[2:4]} and {state[4:6]}. What action should be taken?"
```

2. **LLM for State Representation**:

```
class LLMStateEncoder(nn.Module):
    def __init__(self, llm_model_name):
        super().__init__()
        self.llm = AutoModel.from_pretrained(llm_model_name)
        self.tokenizer = AutoTokenizer.from_pretrained(llm_model_name)

    def forward(self, text_description):
        inputs = self.tokenizer(text_description, return_tensors="pt")
        return self.llm(**inputs).last_hidden_state.mean(dim=1)
```

3. **Memory-Augmented LLM-RL**:

```
class MemoryModule(nn.Module):
    def __init__(self, hidden_size):
        super().__init__()
        self.memory = deque(maxlen=10)
        self.rnn = nn.GRU(hidden_size, hidden_size)

    def forward(self, x):
        # x: current state/observation
        if len(self.memory) > 0:
            memory_tensor = torch.stack(list(self.memory))
            _, hidden = self.rnn(memory_tensor.unsqueeze(1))
            x = x + hidden.squeeze(0)
        self.memory.append(x)
        return x
```

PROF

Practical Considerations

1. **Compute Efficiency**:
    - Use smaller LLMs or distilled versions for real-time applications
    - Cache LLM embeddings when possible
    - Consider quantization for deployment

2. **Training Stability**:
    - Warm-start with imitation learning from LLM suggestions
    - Use KL divergence penalties to stay close to LLM recommendations
    - Implement reward normalization when combining LLM and environment rewards
3. **Evaluation Metrics**:
    - Task performance (traditional RL metrics)
    - Alignment with language instructions
    - Human preference scores