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
```

```python
            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]
```

PROF

```
            # 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",
                        },
```

PROF

```
                        "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)
```

```
    # 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)
```

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.