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
```

PROF

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))
```

PROF

```
        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()

            # 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)
```

PROF

```
    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
```

PROF

```
    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}")
```

PROF

```
    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:

PROF

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.