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)
        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")
```

PROF

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)
```

PROF

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
```

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