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)
        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)
    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.

PROF

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