Actor-Critic Reinforcement Learning for CartPole

Overview

The Actor-Critic method is a reinforcement learning (RL) algorithm that combines the benefits of both policy-based and value-based approaches. It utilizes two neural networks:

- **Actor**: Determines the action to take given a state.
- **Critic**: Evaluates the action taken by estimating the value function.

This dual-network structure enables the model to learn more efficiently by reducing variance in policy gradient estimates.

Implementation

Below is a PyTorch implementation of the Actor-Critic method applied to the CartPole-v1 environment from OpenAI's Gym.

```
import gym
import torch
import torch.nn as nn
import torch.optim as optim

# Initialize environment
env = gym.make("CartPole-v1")
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.n

# Define the Actor-Critic model
class ActorCritic(nn.Module):
    def __init__(self, state_dim, action_dim):
        super(ActorCritic, self).__init__()
        self.actor = nn.Sequential(
            nn.Linear(state_dim, 128),
            nn.ReLU(),
            nn.Linear(128, action_dim),
            nn.Softmax(dim=-1)
        )
        self.critic = nn.Sequential(
            nn.Linear(state_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 1)
        )

    def forward(self, state):
        action_probs = self.actor(state)
        value = self.critic(state)
        return action_probs, value

# Initialize model and optimizer
model = ActorCritic(state_dim, action_dim)
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training function
def train_actor_critic(env, model, optimizer, episodes=500):
    gamma = 0.99  # Discount factor
    for episode in range(episodes):
        state = env.reset()
```

```
        state = torch.FloatTensor(state)
        log_probs = []
        values = []
        rewards = []

        # Generate an episode
        while True:
            action_probs, value = model(state)
            action = torch.multinomial(action_probs, 1).item()
            next_state, reward, done, _ = env.step(action)
            log_probs.append(torch.log(action_probs[action]))
            values.append(value)
            rewards.append(reward)

            state = torch.FloatTensor(next_state)
            if done:
                break

        # Compute returns and losses
        returns = []
        G = 0
        for r in reversed(rewards):
            G = r + gamma * G
            returns.insert(0, G)
        returns = torch.FloatTensor(returns)

        policy_loss = []
        value_loss = []
        for log_prob, value, G in zip(log_probs, values, returns):
            advantage = G - value.item()
            policy_loss.append(-log_prob * advantage)
            value_loss.append((value - G) ** 2)

        optimizer.zero_grad()
        loss = torch.stack(policy_loss).sum() + torch.stack(value_loss).sum()
        loss.backward()
        optimizer.step()

        if (episode + 1) % 50 == 0:
            print(f"Episode {episode+1}/{episodes}, Loss: {loss.item():.4f}")

# Train the model
train_actor_critic(env, model, optimizer)
```

Key Components

- **Actor Network**: Outputs a probability distribution over possible actions given the current state.
- **Critic Network**: Estimates the value function, providing feedback to the actor on the quality of actions taken.
- **Training Loop**: Collects experiences, computes returns, and updates both networks using backpropagation.

Expected Outputs

- **Training Progress**: The console will display the loss at every 50th episode, indicating the model's learning progress.

- **Performance**: Over time, the agent should improve its performance in the CartPole environment, balancing the pole for longer durations.

Use Cases

- **Robotics**: Training robots to perform tasks requiring sequential decision-making.

- **Game AI**: Developing intelligent agents capable of playing complex games.

- **Autonomous Vehicles**: Enabling self-driving cars to make real-time driving decisions.

Advantages

- **Reduced Variance**: Combining policy and value functions helps in reducing the variance of policy gradient estimates.

- **Sample Efficiency**: Actor-Critic methods often require fewer samples to achieve good performance compared to pure policy gradient methods.

Future Enhancements

- **Experience Replay**: Implementing experience replay to break correlation between consecutive samples and improve learning stability.

- **Advanced Architectures**: Exploring more complex neural network architectures for both actor and critic to capture intricate patterns.

- **Hyperparameter Optimization**: Tuning hyperparameters such as learning rates, discount factors, and network sizes to enhance performance.

References

- Actor Critic model to play Cartpole game - GitHub

- Actor-Critic - The A2C Reinforcement Learning Method - GitHub

- Advantage Actor-Critic (A2C) Algorithm Explained and Implemented in PyTorch

- PyTorch program for Cartpole | Reinforcement Learning | Actor-Critic

---

For a visual walkthrough of implementing the Actor-Critic method in PyTorch, you might find the following video helpful:

PyTorch program for Cartpole | Reinforcement Learning | Actor-Critic