Policy-Based Methods (REINFORCE)

Overview

This project implements the **REINFORCE algorithm**, a policy-based reinforcement learning method, using a neural network to directly learn an optimal policy for solving the **CartPole-v1** environment.

Key Features

1. **Environment**:
    - OpenAI Gym's CartPole-v1 environment is used to test the policy.
    - The agent learns to balance a pole on a cart by controlling its movements.
2. **Policy Network**:
    - A neural network maps the state of the environment to a probability distribution over actions.
    - Consists of:
        - Two hidden layers with 24 neurons each and **ReLU activation**.
        - An output layer with softmax activation for action probabilities.
3. **Training**:
    - **Policy Gradient Loss**: The agent optimizes the log-probability of actions weighted by discounted rewards.
    - **Discount Factor (?)**: Accounts for the value of future rewards.
    - **Adam Optimizer**: Updates the policy network's parameters.

How It Works

1. **Initialization**:
    - A policy network is created using TensorFlow.
    - Hyperparameters such as learning rate, discount factor, and number of episodes are defined.
2. **REINFORCE Algorithm**:
    - For each episode:
        1. **Collect Trajectory**:
            - The agent interacts with the environment, selecting actions based on the policy.
            - Records rewards, actions, and log-probabilities of actions.
        2. **Discount Rewards**:
            - Rewards are discounted over time to prioritize earlier actions.
        3. **Policy Update**:
            - Compute the loss using log-probabilities and discounted rewards.
            - Apply gradients to improve the policy.
3. **Execution**:
    - After training, the agent learns to maximize rewards by balancing the pole effectively.

Code Walkthrough

1. **Policy Network**:

```
def create_policy_network():
    model = tf.keras.Sequential([
        layers.Dense(24, activation='relu', input_shape=env.observation_space.shape
        layers.Dense(24, activation='relu'),
        layers.Dense(env.action_space.n, activation='softmax')
    ])
    return model
```

2. **Loss Computation**:

```
def compute_loss(log_probs, rewards, gamma=0.99):
    discounted_rewards = []
    cumulative_reward = 0
    for reward in rewards[::-1]:
        cumulative_reward = reward + cumulative_reward * gamma
        discounted_rewards.insert(0, cumulative_reward)
    loss = -np.sum(log_probs * np.array(discounted_rewards))
    return loss
```

3. **Training Loop**:

```
def reinforce(env, n_episodes=1000):
    for episode in range(n_episodes):
        state = env.reset()
        while not done:
            action_probs = policy_model(state)
            action = np.random.choice(np.arange(env.action_space.n), p=action_probs
            next_state, reward, done, _ = env.step(action)
            rewards.append(reward)
            log_probs.append(np.log(action_probs[0][action]))
            state = next_state
        train_step(np.array(actions), rewards)
```

Advantages

- **Direct Policy Optimization**: Avoids the need for value function approximation.
- **Scalability**: Works well with continuous action spaces and high-dimensional environments.

Future Work

- Implement baseline subtraction to reduce variance in policy gradients.
- Test the REINFORCE algorithm on more complex environments (e.g., LunarLander).
- Compare performance with actor-critic methods.

References

- [OpenAI Gym Documentation]
- [Deep Reinforcement Learning by Sutton & Barto]