Homework#2

(CSE 584)

Abstract: The code trains an agent in a solution of the **CartPole-v1** scenario of the OpenAI Gym using Reinforcement Learning according to the **Policy Gradient approach**. This simple control problem involves applying forces left or right to balance a pole on a cart. This is designed to train an agent by trial and error to learn how to act to get the maximum reward possible: it initializes the environment, samples actions regarding a policy, collects rewards, and updates the policy with information gathered. The approach emphasizes the need for understanding and employing principles of reinforcement learning itself, rather than relying on other libraries for RL-specific tasks.

Probably bound in the notebook, it would include the following:

- Overview of Policy Gradient Approaches could explain how REINFORCE falls into these approaches. It can describe, in theory, the update rule basics for reinforcement learning environments using gradients.
- This sets up a simulation environment within which an agent is allowed to interact and learn the best course of action. This can be setting up an environment using libraries like OpenAI's Gym.
- The neural network representation of the policy function is called Policy Network. This network takes states as input and yields a probability distribution over actions.
- The basic building block of the REINFORCE algorithm is a training loop that runs episodes in the environment. Training loop continues to gather paths of states, actions, and rewards. At each time step, it calculates returns, which is the sum of all future rewards. Then it calculates the policy gradient with gradient ascent and updates the parameters of the policy network.
- By updating the policy network weights through computed gradients according to optimization techniques, including but not limited to Adam or stochastic gradient descent.
- Performance evaluation consists of monitoring the performance of an agent over time; usually, this is represented in terms of success rate or average payment per episode.

Core section of the Reinforcement Learning Implementation:

1. Imports and Setup:

```python
# Importing the gym library for creating and interacting with environments
import gym

# Importing numpy for numerical operations
import numpy as np

# Importing PyTorch for building and training neural networks
import torch

# Importing neural network module from PyTorch
```

```
import torch.nn as nn

# Importing optimization module from PyTorch
import torch.optim as optim
```

In this section, the libraries are needed to be imported in order to set up the environment, do the numerical computations, and create and train the neural network model will be imported.

2. Policy Network Definition:

```
# Define the PolicyNetwork class, inheriting from nn.Module
class PolicyNetwork(nn.Module):
    # Initialize the network with input and output sizes
    def __init__(self, input_size, output_size):
        # Call the constructor of the parent class (nn.Module)
        super(PolicyNetwork, self).__init__()
        # Create the first fully connected layer with 128 hidden units
        self.fc1 = nn.Linear(input_size, 128)
        # Create the output layer with size matching the action space
        self.fc2 = nn.Linear(128, output_size)

    # Define the forward pass of the network
    def forward(self, x):
        # Apply ReLU activation function to the output of the first layer
        x = torch.relu(self.fc1(x))
        # Apply softmax to the output layer to get action probabilities
        x = torch.softmax(self.fc2(x), dim=-1)
        # Return the action probabilities
        return x
```

Defines a simple feedforward neural network used to approximate the policy. The network has two layers: an output layer with softmax activation and a hidden layer with ReLU activation.

3. Action Selection Function:

```
# Define a function to select an action based on the current policy and state
def select_action(policy_net, state):
    # Convert the state from a numpy array to a PyTorch tensor, add a batch
dimension
    state = torch.from_numpy(state).float().unsqueeze(0)
    # Pass the state through the policy network to get action probabilities
    probs = policy_net(state)
    # Sample an action based on the probabilities using numpy's random choice
    action = np.random.choice(len(probs[0]), p=probs.detach().numpy()[0])
    # Return the selected action
    return action
```

It samples from the probability distribution that the policy network outputs given the current state to select an action.

4. Environment Initialization:

```
# Create CartPole environment
env = gym.make('CartPole-v1')
```

Initializes the CartPole environment, within which the agent will act. The environment is that, which provides the interface necessary to execute episodes.

5. Agent Initialization:

```
# Initialize the policy network with input size 4 (state space dimension) and
output size 2 (action space dimension)
policy_net = PolicyNetwork(input_size=4, output_size=2)

# Set up the Adam optimizer for the policy network parameters with a learning rate
of 0.01
optimizer = optim.Adam(policy_net.parameters(), lr=0.01)
```

Initializes the agent with instantiation of PolicyNetwork and an optimizer: basically, Adam method to update weights in training.

6. Training Loop:

6.1 Episode Loop:

```
# Start the main training loop, running for 1000 episodes
for episode in range(1000):
    # Reset the environment and get the initial state
    state = env.reset()
    # Initialize an empty list to store rewards for this episode
    rewards = []

    # Start the episode loop, maximum of 100 steps per episode
    for t in range(100):
        # Select an action based on the current state using the policy network
        action = select_action(policy_net, state)
        # Take the selected action in the environment, receiving the next state,
reward, and done flag
        next_state, reward, done, _ = env.step(action)
        # Append the received reward to the rewards list
        rewards.append(reward)
        # Update the current state to the new state
        state = next_state

        # Check if the episode has finished (pole has fallen or max steps reached)
        if done:
            # Exit the episode loop if the episode is done
            break
```

- **Episode Initialization:** Environment reset at each episode.
- **Step Loop:** Loop through steps in episode, selects actions by current policy, gather rewards

6.2 Policy update

```
# Calculate the total reward (return) for the episode by summing all rewards
R = sum(rewards)

# Calculate the loss using the REINFORCE algorithm formula
# This is the negative log probability of the chosen action multiplied by the total
reward
loss = -torch.log(probs[action]) * R

# Zero out the gradients of the optimizer
# This is necessary before computing gradients for a new batch
optimizer.zero_grad()

# Compute the gradients of the loss with respect to the network parameters
loss.backward()

# Update the network parameters using the computed gradients
optimizer.step()
```

- **Reward Calculation:** Calculate sum of rewards gathered in episode.
- **Loss Calculation:** It utilizes the policy gradient methods to compute a loss concerning actions taken along with their rewards.
- **Backpropagation and Optimization:** The weights in the policy network are updated using backpropagation and gradient descent.

7. Training Execution

```
# Call train function to start training process
train()
```

This line starts the training process by calling the function train, which will execute all steps previously defined in order.

Github Link: https://github.com/udacity/deep-reinforcement-learning/tree/master/reinforce