HOMEWORK 2 - CSE 584

Mega Sri Shyam Bollini
PSU ID - 991264286
mpb6512@psu.edu

1) Abstract:

This code implements the Advantage Actor-Critic (A2C) algorithm, a popular reinforcement learning approach that combines elements of policy-based and value-based methods. The A2C algorithm addresses the CartPole-v1 environment from OpenAI’s Gym, where the agent's objective is to balance a pole on a moving cart. The agent is designed using an actor-critic architecture, which involves two separate neural networks: an actor that determines the optimal action policy, and a critic that evaluates the value of a given state to estimate the expected return.

The actor network outputs a probability distribution over possible actions, using softmax activation, while the critic network predicts the value of the state, helping to calculate the advantage function, which measures how much better or worse an action performs compared to the expected value of that state. The training process alternates between updating the actor to maximize the advantage function and updating the critic to minimize the mean squared error between predicted and actual state values. The neural networks are trained with different learning rates to optimize performance: the actor uses a lower learning rate (0.001) to ensure stable policy updates, while the critic uses a higher learning rate (0.005) to provide accurate state value estimates.

The code includes provisions for both training and evaluating the model, with additional functionality for saving and loading model weights. During each episode, the agent stochastically selects actions based on the policy distribution, receives rewards, and trains iteratively to improve policy accuracy and state value estimates. The environment is reset at the start of each episode, and the training loop continues until the agent consistently achieves a high score, indicating mastery of the task. If the average score over the last ten episodes exceeds 490, the training terminates early. Regular model checkpoints ensure that progress is saved, and the model can be reloaded for future use.

This implementation of A2C demonstrates how actor-critic methods can efficiently solve continuous control problems, leveraging neural networks for both policy approximation and value estimation, with hyperparameters tuned to the CartPole-v1 environment’s characteristics.

2) Code Analysis:

```
# Importing necessary libraries
import sys
```

```
import gym
import pylab
import numpy as np
from keras.layers import Dense
from keras.models import Sequential
from keras.optimizers import Adam

# Number of episodes for training
EPISODES = 1000

# A2C(Advantage Actor-Critic) agent for the CartPole environment
class A2CAgent:
    def __init__(self, state_size, action_size):
        # Rendering flag: set to True to visualize the environment
        self.render = False
        # Load model flag: set to True to load saved weights
        self.load_model = False
        # State and action sizes
        self.state_size = state_size
        self.action_size = action_size
        self.value_size = 1  # Output size of the critic

        # Hyperparameters for the Policy Gradient
        self.discount_factor = 0.99  # Discount factor for future rewards
        self.actor_lr = 0.001  # Learning rate for the actor network
        self.critic_lr = 0.005  # Learning rate for the critic network

        # Create models for the actor and critic networks
        self.actor = self.build_actor()  # Actor model for policy approximation
        self.critic = self.build_critic()  # Critic model for value approximation

        # Load saved models if the load_model flag is set
        if self.load_model:
            self.actor.load_weights("./save_model/cartpole_actor.h5")
            self.critic.load_weights("./save_model/cartpole_critic.h5")

    # Build the actor model for policy approximation
    def build_actor(self):
        actor = Sequential()
        # Input layer with 24 neurons, ReLU activation
        actor.add(Dense(24, input_dim=self.state_size, activation='relu',
                        kernel_initializer='he_uniform'))
        # Output layer with softmax activation for probability distribution of actions
        actor.add(Dense(self.action_size, activation='softmax',
```

```
                kernel_initializer='he_uniform'))
    actor.summary()  # Display the model summary
    # Compile the actor model using categorical crossentropy loss and Adam optimizer
    actor.compile(loss='categorical_crossentropy',
              optimizer=Adam(lr=self.actor_lr))
    return actor

# Build the critic model for value approximation
def build_critic(self):
    critic = Sequential()
    # Input layer with 24 neurons, ReLU activation
    critic.add(Dense(24, input_dim=self.state_size, activation='relu',
                kernel_initializer='he_uniform'))
    # Output layer with linear activation for value prediction
    critic.add(Dense(self.value_size, activation='linear',
                kernel_initializer='he_uniform'))
    critic.summary()  # Display the model summary
    # Compile the critic model using mean squared error (MSE) loss and Adam optimizer
    critic.compile(loss="mse", optimizer=Adam(lr=self.critic_lr))
    return critic

# Get action based on policy network output
def get_action(self, state):
    policy = self.actor.predict(state, batch_size=1).flatten()
    # Select an action stochastically based on the policy probabilities
    return np.random.choice(self.action_size, 1, p=policy)[0]

# Train the actor and critic models
def train_model(self, state, action, reward, next_state, done):
    target = np.zeros((1, self.value_size))  # Target value for critic
    advantages = np.zeros((1, self.action_size))  # Advantage values for actor

    # Predict value for the current and next states
    value = self.critic.predict(state)[0]
    next_value = self.critic.predict(next_state)[0]

    # Compute advantages and targets based on whether the episode is done
    if done:
        advantages[0][action] = reward - value
        target[0][0] = reward
    else:
        advantages[0][action] = reward + self.discount_factor * next_value - value
        target[0][0] = reward + self.discount_factor * next_value
```

```
        # Update the actor and critic networks
        self.actor.fit(state, advantages, epochs=1, verbose=0)
        self.critic.fit(state, target, epochs=1, verbose=0)

if __name__ == "__main__":
    env = gym.make('CartPole-v1')  # Create the CartPole-v1 environment
    state_size = env.observation_space.shape[0]  # State size
    action_size = env.action_space.n  # Action size

    agent = A2CAgent(state_size, action_size)  # Initialize the A2C agent

    scores, episodes = [], []  # Lists for scores and episode numbers

    # Main training loop
    for e in range(EPISODES):
        done = False  # Episode termination flag
        score = 0  # Episode score
        state = env.reset()  # Reset the environment at the start of each episode
        state = np.reshape(state, [1, state_size])  # Reshape state for NN input

        while not done:
            if agent.render:
                env.render()  # Render the environment if the render flag is set

            action = agent.get_action(state)  # Get action from the policy network
            next_state, reward, done, info = env.step(action)  # Take a step in the environment
            next_state = np.reshape(next_state, [1, state_size])  # Reshape next state

            # Assign penalty if the episode ends early
            reward = reward if not done or score == 499 else -100

            agent.train_model(state, action, reward, next_state, done)  # Train the agent

            score += reward  # Update episode score
            state = next_state  # Update current state

            if done:
                score = score if score == 500.0 else score + 100
                scores.append(score)
                episodes.append(e)
                pylab.plot(episodes, scores, 'b')  # Plot scores
                pylab.savefig("./save_graph/cartpole_a2c.png")
                print("episode:", e, "  score:", score)
```

```
        # Stop training if the average score of last 10 episodes is greater than 490
        if np.mean(scores[-min(10, len(scores)):]) > 490:
            sys.exit()

    # Save models at every 50th episode
    if e % 50 == 0:
        agent.actor.save_weights("./save_model/cartpole_actor.h5")
        agent.critic.save_weights("./save_model/cartpole_critic.h5")
```