Reinforcement Learning with Semi-Supervision

Project Overview

This project explores the application of **Reinforcement Learning (RL)** combined with **Semi-Supervised Learning** to enhance the performance of a Q-learning agent in an environment, specifically the **CartPole-v1** environment from OpenAI's Gym. In this setup, we utilize **Label Spreading**, a semi-supervised learning method, to propagate pseudo-labels for the agent’s actions based on observed rewards. This method leverages both labeled and unlabeled data for improved model performance, enabling the agent to learn more effectively in situations with sparse labels.

Why We Use This Particular Model

Reinforcement learning traditionally relies on explicit rewards for learning optimal policies. In this project, we introduce **semi-supervised learning** to address situations where rewards are sparse or difficult to obtain for all possible actions. By applying **pseudo-labeling** via **Label Spreading**, the agent can infer additional labels for previously unseen state-action pairs, facilitating faster learning and better exploration of the environment.

By combining RL with semi-supervised learning, we aim to leverage a small amount of labeled data and augment it with pseudo-labeled data, thus improving the model's ability to generalize and accelerate the learning process.

##**Prerequisites**

Required Libraries

The following libraries are required to run the project:

- `numpy`
- `gym`
- `sklearn`
- `matplotlib`
- `collections`
- `random`

Installation

To install the required libraries, run the following command:

```
pip install numpy gym scikit-learn matplotlib
```

Files Included

This project includes the following files:

1. **main.py**: The primary Python file containing the implementation of the Q-learning agent with semi-supervised learning.
2. **requirements.txt**: A text file listing the necessary dependencies to install via pip.
3. **README.md**: This file, containing the project overview, description, and instructions for use.

Code Description

1. Importing Required Libraries

The project begins by importing the necessary libraries to handle RL, semi-supervised learning, and the required utilities:

```
import numpy as np
import gym
from sklearn.semi_supervised import LabelSpreading
from sklearn.metrics import accuracy_score
from collections import deque
import random
```

- `numpy` is used for numerical operations.
- `gym` is used to create the CartPole environment.
- `LabelSpreading` from `sklearn` handles the semi-supervised learning.
- `deque` and `random` are used for managing state-action pairs and action selection.

2. Initializing the Gym Environment

Next, we initialize the CartPole environment and set up training parameters:

```
env = gym.make('CartPole-v1')  # Example environment for RL

n_episodes = 1000
max_timesteps = 200
```

- `gym.make('CartPole-v1')` sets up the CartPole environment where the agent needs to balance a pole on a moving cart.
- `n_episodes` defines the number of training episodes, while `max_timesteps` sets the maximum steps per episode.

3. Preprocessing States

The following function is defined to preprocess the state, but in this case, it is just a placeholder function that converts the state into a numpy array:

```
def preprocess_state(state):
    return np.array(state)
```

This is useful in case we want to apply normalization or transformations to the states in the future.

4. Defining the QLearningAgent Class

The heart of the RL agent is the `QLearningAgent` class. It contains methods for Q-value management, action selection, and Q-value updates.

Initialization and Q-value Management

```
class QLearningAgent:
    def __init__(self, n_actions, gamma=0.99, alpha=0.1, epsilon=0.1):
        self.n_actions = n_actions
        self.gamma = gamma
        self.alpha = alpha
        self.epsilon = epsilon
        self.Q = {}

    def get_q(self, state, action):
        return self.Q.get((tuple(state), action), 0)
```

- The agent is initialized with `n_actions`, `gamma` (discount factor), `alpha` (learning rate), and `epsilon` (for epsilon-greedy policy).

- `get_q` retrieves the Q-value for a given state-action pair, defaulting to 0 if not yet encountered.

Q-value Update

```
    def update_q(self, state, action, reward, next_state, done):
        best_next_action = np.argmax([self.get_q(next_state, a) for a in range(self.n_ac
        td_target = reward + (self.gamma * self.get_q(next_state, best_next_action) * (1
        td_error = td_target - self.get_q(state, action)
        self.Q[(tuple(state), action)] = self.get_q(state, action) + self.alpha * td_err
```

- The `update_q` function updates the Q-value for a state-action pair using the **Temporal Difference (TD) error**. The agent learns by adjusting the Q-values toward the best possible next action.

Action Selection

```
    def select_action(self, state):
        if random.random() < self.epsilon:
            return np.random.choice(self.n_actions)
        else:
            q_values = [self.get_q(state, action) for action in range(self.n_actions)]
            return np.argmax(q_values)
```

- The `select_action` method uses an epsilon-greedy strategy to choose an action. With probability `epsilon`, the agent explores randomly; otherwise, it exploits the best-known action.

5. Semi-Supervised Learning with Label Spreading

To handle unlabeled data, **Label Spreading** is used for pseudo-labeling:

```
label_spread_model = LabelSpreading(kernel='rbf', gamma=20)

def apply_semi_supervised_learning(state_action_pairs, rewards):
    pseudo_labels = np.array([1 if r > 0 else 0 for r in rewards])  # 1 for positive rew
    label_spread_model.fit(state_action_pairs, pseudo_labels)
    return label_spread_model.predict(state_action_pairs)
```

- `LabelSpreading` is initialized and fitted on state-action pairs with rewards as labels. The model predicts pseudo-labels (1 for positive rewards, 0 for negative) for state-action pairs that do not have labels.

6. Training Loop

The core training loop consists of episodes where the agent interacts with the environment and updates the Q-values:

```
agent = QLearningAgent(n_actions=env.action_space.n)
state_action_pairs = []
rewards = []

for episode in range(n_episodes):
    state = env.reset()
    total_reward = 0
    done = False

    for timestep in range(max_timesteps):
        action = agent.select_action(state)
        next_state, reward, done, _ = env.step(action)

        # Store state-action pairs and rewards
        state_action_pairs.append(np.concatenate([state, [action]]))
```

```
        rewards.append(reward)

        # Update the Q-table with the reward
        agent.update_q(state, action, reward, next_state, done)

        state = next_state
        total_reward += reward

        if done:
            break

    # Apply semi-supervised learning (pseudo-labeling) periodically
    if episode % 10 == 0:
        pseudo_labels = apply_semi_supervised_learning(np.array(state_action_pairs), np
        print(f"Episode {episode}, Pseudo-labeling applied.")

    print(f"Episode {episode}, Total reward: {total_reward}")
```

- The agent interacts with the environment and collects state-action pairs and rewards. Every 10 episodes, semi-supervised learning is applied to generate pseudo-labels, helping the agent learn more effectively.

7. Evaluating the Agent

After training, the agent is evaluated over 10 test episodes:

```
total_rewards = 0
for _ in range(10):
    state = env.reset()
    done = False
    while not done:
        action = agent.select_action(state)
        state, reward, done, _ = env.step(action)
        total_rewards += reward

print(f"Average Reward over 10 test episodes: {total_rewards / 10}")
```

- The agent's performance is measured by running 10 test episodes, where the total reward is accumulated and averaged.

8. Visualizing Performance

Lastly, the agent's performance is plotted over time using `matplotlib`:

```
import matplotlib.pyplot as plt

plt.plot(range(n_episodes), total_rewards)
plt.title("Agent's Performance over Episodes")
plt.xlabel("Episodes")
plt.ylabel("Total Reward")
plt.show()
```

- The performance plot helps visualize how the agent's reward improves over the episodes.

Expected Outputs

The following outputs are expected when running the project:

1. **Console Output**: A printout showing the total reward for each episode, along with messages indicating when pseudo-labeling has been applied. Example:

```
 Episode 10, Pseudo-labeling applied.
Episode 10, Total reward: 120
```

2. **Visualization**: A plot showing the agent's total reward over episodes, visualizing the improvement in the agent's performance as the episodes progress.

3. **Average Reward**: The average reward over a set of test episodes is printed at the end of training, showcasing the agent's overall performance. Example:

```
Average Reward over 10 test episodes: 250
```

Use Cases

- **Robotics**: The agent can be adapted to learn control policies for robots in environments where rewards are sparse or hard to obtain.
- **Gaming AI**: Similar methods can be applied to develop intelligent agents for video games that learn through reinforcement with semi-supervised learning.
- **Autonomous Vehicles**: The project can be extended to simulate driving policies for self-driving cars in partially labeled environments.
- **Financial Modeling**: This approach can be used to optimize trading strategies where actions (buy, sell, hold) need to be learned based on limited reward feedback.

Future Enhancements

1. **Integrating Advanced RL Algorithms

: Further work can involve integrating more advanced RL algorithms like **Deep Q-Learning or **Proximal Policy Optimization** (PPO). 2. **Exploration-Exploitation Trade-offs**: The exploration strategy can be enhanced by integrating methods such as **UCB (Upper Confidence Bound)** or **Thompson Sampling** for better action selection. 3. **Semi-Supervised Learning Improvements**: Experiment with other semi-supervised techniques like **Self-Training** and **Co-training** for more robust learning.

This structured approach allows for an easy understanding of the project, from high-level overview to code breakdown. Each section is detailed with explanations and code segments that guide users through the core components of the project.