Introduction to Reinforcement Learning: Tutorial and Final Project

Introduction

Welcome to your introduction to Reinforcement Learning (RL)! This tutorial will guide you through understanding RL concepts through hands-on experiments, culminating in a final project where you'll create your own experiments.

What is Reinforcement Learning?

Reinforcement Learning is a type of machine learning where an agent learns to make decisions by interacting with an environment. Unlike supervised learning where we have labeled data, in RL the agent learns through trial and error, receiving rewards or penalties for its actions. Think of how a child learns to ride a bicycle - they try different movements, fall a few times, but gradually learn what actions lead to staying balanced. For a primer, see the following link!

Real-World Applications

RL is used in many exciting real-world applications:

- Robotics: Teaching robots to walk, grasp objects, or navigate environments
- Game AI: DeepMind's AlphaGo beating world champions at Go
- Resource Management: Data center cooling optimization (Google)
- Autonomous Vehicles: Self-driving cars learning to navigate traffic
- Finance: Trading algorithms adapting to market conditions
- Healthcare: Personalized treatment recommendations

Tutorial Goals

In this tutorial, you will: 1. Understand basic RL concepts through hands-on experiments 2. See how different parameters affect learning behavior 3. Visualize how agents learn over time 4. Create your own experiments for the final project

Creating a Virtual Environment

Using Command Line

1. Open your terminal/command prompt

2. Navigate to your project directory:

   ```
   cd path/to/your/project
   ```

3. Create a virtual environment:

   - Windows: `python -m venv venv`
   - Mac/Linux: `python3 -m venv venv`

4. Activate the virtual environment:

   - Windows: `venv\Scripts\activate`
   - Mac/Linux: `source venv/bin/activate`

Using VSCode

1. Open your project folder in VSCode
2. Press `Ctrl+Shift+P` (Windows) or `Cmd+Shift+P` (Mac)
3. Type “Python: Select Interpreter”
4. Click “+ Enter interpreter path”
5. Create a new virtual environment
6. Select the new virtual environment

Using PyCharm

1. Open your project in PyCharm
2. Go to File → Settings → Project → Python Interpreter
3. Click the gear icon → Add
4. Choose “New environment” and click OK
5. PyCharm will create and configure the virtual environment

Installing Required Packages

First, let’s install the required packages. After activating your virtual environment:

```
pip install "gymnasium[classic-control]" numpy matplotlib
```

Part 1: Understanding the Environment

Let’s start by watching how our environment (CartPole) works. This code shows a random agent. Copy the code and run it locally.

```
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt

from time import sleep

env = gym.make('CartPole-v1', render_mode='human')
state, _ = env.reset()

# Watch random actions
for _ in range(1000):
    action = env.action_space.sample()  # Random action
    state, reward, terminated, truncated, _ = env.step(action)
    if terminated or truncated:
        print('The cartpole has tipped over!')
        sleep(2)
        state, _ = env.reset()
env.close()
```

Watch the terminal as your code runs to see how often the cartpole falls! We have included a sleep which pauses the program for 2 seconds to illustrate how often the cartpol falls over (every pause indicates a fall).

Think About It: What happens when you run this code? Why does the pole fall so quickly?

Question 1: In 50-75 words, detail what happens when you run this code and explain why the pole falls so quickly.

Part 2: Your First RL Agent

Our first approach is to try the battle-tested method, Q-learning. Q-learning is a fundamental reinforcement learning algorithm introduced by Watkins & Dayan in their 1992 paper “Q-learning.” The algorithm learns to make decisions by maintaining a table of state-action values (Q-values) that estimate the expected future reward for taking each action in each state. For a deeper understanding, refer to the original paper.

Now let’s create a Q-learning agent. Don’t worry about understanding all the code yet - we’ll experiment with it first!

```python
class QLearningAgent:
    def __init__(self, learning_rate=0.1, discount_factor=0.95, epsilon=1.0,
        epsilon_decay=0.995):
        self.q_table = {}
        self.lr = learning_rate          # How quickly the agent learns from new
        experiences
        self.gamma = discount_factor      # How much the agent values future rewards
        self.epsilon = epsilon           # How often the agent explores vs exploits
        self.epsilon_decay = epsilon_decay  # How quickly exploration decreases

    def discretize_state(self, state):
        # Convert continuous state to discrete state
        discrete_state = tuple(np.round(state, decimals=1))
        return discrete_state

    def get_action(self, state):
        discrete_state = self.discretize_state(state)

        # Exploration: random action
        if np.random.random() < self.epsilon:
            return np.random.choice([0, 1])

        # Exploitation: best known action
        if discrete_state not in self.q_table:
            self.q_table[discrete_state] = [0, 0]
        return np.argmax(self.q_table[discrete_state])

    def learn(self, state, action, reward, next_state, done):
        discrete_state = self.discretize_state(state)
        discrete_next_state = self.discretize_state(next_state)

        # Initialize Q-values if state is new
        if discrete_state not in self.q_table:
            self.q_table[discrete_state] = [0, 0]
        if discrete_next_state not in self.q_table:
            self.q_table[discrete_next_state] = [0, 0]

        # Q-learning update
        old_value = self.q_table[discrete_state][action]
        next_max = np.max(self.q_table[discrete_next_state])
        new_value = (1 - self.lr) * old_value + self.lr * (reward + self.gamma * next_max)
        self.q_table[discrete_state][action] = new_value

        # Decay exploration rate
        if done:
            self.epsilon *= self.epsilon_decay

def train_and_plot(agent, episodes=100):
    env = gym.make('CartPole-v1')
    rewards = []

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

        while not done:
            action = agent.get_action(state)
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            agent.learn(state, action, reward, next_state, done)
            state = next_state
            total_reward += reward

        rewards.append(total_reward)
        if episode % 10 == 0:
            print(f"Episode {episode}, Score: {total_reward}, Epsilon:
{agent.epsilon:.2f}")

    env.close()
    return rewards

# Train the agent with default parameters
agent = QLearningAgent()
rewards = train_and_plot(agent)

# Plot the results
plt.plot(rewards)
plt.title('Learning Progress')
plt.xlabel('Episode')
plt.ylabel('Score')
plt.show()
```

Understanding the Parameters

Let's break down what each parameter does:

1. **Learning Rate (lr)**: How quickly the agent learns from new experiences
   - High (e.g., 0.5): Learn a lot from each experience, but might be unstable
   - Low (e.g., 0.1): Learn slowly but more steadily
2. **Epsilon (ε)**: How often the agent explores vs exploits
   - Starts high (1.0): Agent tries random actions (i.e., tries to learn a new strategy– exploration)
   - Decreases over time: Agent gradually relies more on learned experience (i.e., stays close to the known strategy– exploitation)
3. **Epsilon Decay**: How quickly exploration decreases
   - Fast decay (e.g., 0.95): Quickly switch to exploitation
   - Slow decay (e.g., 0.995): Keep exploring longer
4. **Discount Factor (gamma)**: How much the agent values future rewards
   - High (e.g., 0.99): Care a lot about future rewards
   - Low (e.g., 0.5): Focus more on immediate rewards

Run the snippet above and examine the results.

Understanding the Results

Looking at the plot from your training, you should notice: 1. Initial Performance: The scores likely start low as the agent is mostly exploring 2. Learning Progress: You should see an upward trend as the agent learns better strategies 3. Variability: The scores probably fluctuate due to the balance between exploration and exploitation 4. Final Performance: By the end, the agent should achieve somewhat higher scores consistently

Think About It: - Why does the learning curve show ups and downs? - How does the exploration rate (epsilon) affect the variability in scores? - What might cause the agent to perform worse after performing well?

Question 2: In 100-150 words, answer the questions above.

Optional: Watch Your Trained Agent!

After training, you can watch how your agent performs:

```
env = gym.make('CartPole-v1', render_mode='human')
state, _ = env.reset()
done = False

while not done:
    action = agent.get_action(state)
    state, reward, terminated, truncated, _ = env.step(action)
    done = terminated or truncated
env.close()
```

Note the performance is not expected to be good. The goal is to train an agent that can solve our task; however, this takes a lot of experimentation!

Part 3: Experimenting with Parameters

Let's try different parameter combinations to see how they affect learning:

```
# Create three different agents
agent1 = QLearningAgent(learning_rate=0.1, epsilon=1.0, epsilon_decay=0.995)  # Default
agent2 = QLearningAgent(learning_rate=0.2, epsilon=1.0, epsilon_decay=0.99)   # Fast
        learning
agent3 = QLearningAgent(learning_rate=0.05, epsilon=1.0, epsilon_decay=0.998) # Slow
        learning

# Function to train multiple agents
def train_multiple_agents(agents, episodes=100):
    results = {}

    for i, agent in enumerate(agents, 1):
        print(f"\nTraining Agent {i}")
        rewards = train_and_plot(agent)
        results[f"Agent {i}"] = rewards

    return results

# Train all agents
agents = [agent1, agent2, agent3]
results = train_multiple_agents(agents)

# Plot comparison
plt.figure(figsize=(10, 5))
for agent_name, rewards in results.items():
    plt.plot(rewards, label=agent_name)
plt.title('Learning Progress Comparison')
plt.xlabel('Episode')
plt.ylabel('Score')
plt.legend()
plt.show()
```

Think About It: - Which agent learned better? Why? - What happens when you change the parameters? - How does the learning curve change with different parameters?

Question 3: In 100-150 words, answer the questions above.

Final Project: Design Your Own Experiments

Now that you understand how the parameters affect learning, it's time to create your own experiments!

Project Requirements

Part 1: Parameter Experimentation (30 points)

Create three different agents by changing these parameters: - Learning rate (lr) - Epsilon (exploration rate) - Epsilon decay rate

For example:

```
# Create three different agents with your chosen parameters
my_agent1 = QLearningAgent(learning_rate=0.1, epsilon=1.0, epsilon_decay=0.995)
my_agent2 = QLearningAgent(learning_rate=0.2, epsilon=1.0, epsilon_decay=0.99)
my_agent3 = QLearningAgent(learning_rate=0.05, epsilon=1.0, epsilon_decay=0.998)
```

Part 2: Analysis (30 points)

Train each agent for 100 episodes and answer these questions in a short write-up (about 200 words):

1. Which agent performed the best? How do you know?
2. What happened when you changed the learning rate?
3. What happened when you changed the epsilon decay rate?
4. If you were to create another agent, what parameters would you choose and why?

Include two plots in your analysis: 1. A learning curve (episode vs. score) comparing all three agents 2. A plot showing how epsilon changes over time for each agent

Bonus: Solving the Task (+10 points)

The CartPole environment is considered "solved" when the agent achieves an average score of 195.0 over 100 consecutive episodes. Here are some strategies to try:

1. Algorithm Improvements
   - Try implementing a Deep Q-Network (DQN) using PyTorch or TensorFlow
   - Add experience replay to reuse important experiences
2. Extended Training
   - Increase the number of training episodes (try 500 or 1000)
   - Use a slower epsilon decay to maintain exploration longer
   - Consider implementing a minimum epsilon value (e.g., 0.01) to maintain some exploration
3. State Space Modifications
   - Experiment with different discretization schemes
   - Try using more decimal places in state discretization
   - Consider normalizing the state values
4. Parameter Tuning Tips
   - Use a higher discount factor ($\gamma = 0.99$) to value future rewards more
   - Try a learning rate schedule that decreases over time
   - Experiment with reward shaping (e.g., giving larger rewards for staying balanced longer)

Remember to document which approaches you tried and their effects on performance. Even if you don't fully solve the environment, a thorough analysis of your attempts and their outcomes can earn you the bonus points!

Submission Requirements

1. Your Python code file containing:

    - The creation of three different agents
    - Training code
    - Plotting code
2. A PDF or text file containing:
    - Your analysis answering the questions from the tutorial and Part 2
    - The two required plots
    - A brief explanation of why you chose your parameter values

Grading Criteria

- Tutorial questions (40%)
- Successful creation and training of three different agents (30%)
- Analysis and explanation of results (30%)

Tips for Success

- Start with small changes to the parameters
- Make sure to save your plots
- Take notes about what you observe during training
- Don't worry about getting "perfect" results - focus on understanding how the changes affect learning

Due Date

Submit your code file and analysis as a zip file through canvas by February 13.

Good luck! Remember, the goal is to experiment and learn - there are no "wrong" parameter choices as long as you can explain your reasoning!