Homework - II (Reinforcement Learning)

Github Link: basic_reinforcement_learning/tutorial2 at master · vmayoral/basic_reinforcement_learning

I. Code Overview: Purpose and Process

1. Environment and Structure

The code organizes the environment in a grid world where a grid of `Cell` objects makes up the environment. Every cell might be of one of several kinds of terrains or obstacles, such as cliffs, walls, starting points, goals, or open space. A configuration with a map, for example, `cliff.txt` or `barrier2.txt`, makes modification of the layout of the environment very easy. Agents navigate this grid following their defined directions of movement and interaction rules or trying to avoid others in some problems. Modular options in display are supported: graphical displays with either `Tkinter` or `Pygame`, and a `DummyDisplay` for log-based, non-graphical analysis. It does so with modularized observation of agent actions and environment states in real-time or via logs saved for such action/state traces.

2. Core Components

The basic building blocks within the code are classes: `Cell`, `Agent`, and `World`. `Cell` objects are the basic units in the grid; they could represent different elements of the environment based on the configuration of the environment in which the agent exists, such as cliffs, walls, or goals. The Agent class defines methods to move, such as turnLeft or goForward, and gives access to neighbors, such as leftCell or aheadCell. Agents are assigned goals, e.g., find the shortest path to the goals while avoiding obstacles. The World class is in charge of the setting of the grid, placing agents, and updating the environment after every step of the simulation. It also controls the interaction between agent-cell and refreshes the visual display so that immediate feedback on agent progress is accomplished. Regarding flow, the grid initializes cells and agents given some configuration; every step of the simulation allows agents to compute and execute an action based on what surrounds them, given movement and target-seeking rules. The display would continuously refresh the advancement that each agent makes within the environment.

3. Reinforcement Learning Algorithms

Here, two of the most famous reinforcement learning algorithms were implemented, Q-learning and SARSA. Methods that have been integrated into various agents to explore and learn the best actions to perform in the grid environment. In the Q-learning setup, as represented in `Cliff_q.py` and `Q_learning.py`, an agent will learn how to avoid cliffs and obstacles while trying to reach a goal. The agent receives a small penalty for each step (-1), a large penalty if it steps off a cliff (-100), and a positive reward for reaching the goal (50). Extensive pretraining-for instance, 100,000 iterations-allows Q-learning to optimize behavior of the agent by updating Q-values for every state-action pair based on expected rewards and to prepare for real-time simulation. In contrast, the SARSA algorithm is a conservative learner: it updates Q-values based on what action will actually be taken in the next state. It has been implemented in `Cliff_s.py` and `sarsa.py`. Thus, this agent learns to navigate in stable states close to cliffs.

4. Specialized Environments and Special Use Cases

Code is also provided for the specialized environments in which these reinforcement learning methods will be tested. In the Cliff World, agents learn to reach a goal without falling off cliffs. Q-learning and SARSA have been done for comparison. Here, Q-learning does an optimization in such a way that the choice of actions is independent of the next chosen action. On the contrary, SARSA relies on the action taken. Other examples include Mouse and Cat: A mouse agent, driven by Q-learning, tries to achieve cheese while avoiding a cat pursuit. First, the mouse receives a reward for reaching the cheese and a penalty for getting caught by the cat, with a slight penalty for each regular step. To make things a bit more interesting, the cat actively chases the mouse, making the task truly difficult. The Q-learning algorithm for the mouse will be of a gradual decreasing exploration rate, going from exploration moves to exploiting learned strategies so it will be fine-tuning its path to the cheese while avoiding the cat.

5. Moduly Q-Learning and SARSA Classes

Q-learning and SARSA classes in Q_learning.py and sarsa.py, respectively, are reusable classes for these algorithms. They store and update Q-values concerning state-action pairs based on rewards received. Their applications are made flexible enough for different RL scenarios within the environment using epsilon-greedy strategies to balance exploration and exploitation.

6. Execution and Training Process

Execution and training will finally start to set up each environment with agents, followed by the long training loop that allows agents to learn optimal behaviors. Both Q-learning- and SARSA-trained agents usually train for hundreds of thousands of iterations, gradually decreasing exploration as their strategies get refined. After this pre-training, agents go into real-time simulation where they apply learned actions based on their policies. The following graphics output shows how the agents move toward objectives—like cheese or goals—or go away from threats, such as cliffs and the cat.

II. Core RL Implementation: Key Functions with Line-by-Line Comments

qlearn.py:

```python
import random

# Q-Learning agent class
class QLearn:
    # Initialize the Q-Learning agent with actions, epsilon, alpha, and gamma
parameters
    def __init__(self, actions, epsilon=0.1, alpha=0.2, gamma=0.9):
        self.q = {}  # Dictionary to store Q-values for state-action pairs
        self.epsilon = epsilon  # Exploration rate for epsilon-greedy policy
        self.alpha = alpha  # Learning rate
        self.gamma = gamma  # Discount factor
        self.actions = actions  # List of possible actions

    # Method to get the Q-value for a given (state, action) pair; returns 0.0 if it doesn't exist
    def getQ(self, state, action):
        return self.q.get((state, action), 0.0)

    # Update Q-value for (state, action) pair using the reward and expected future value
    def learnQ(self, state, action, reward, value):
        oldv = self.q.get((state, action), None)
        if oldv is None:  # If Q-value doesn't exist, initialize it with the reward
            self.q[(state, action)] = reward
        else:  # Otherwise, update Q-value using the Q-learning formula
            self.q[(state, action)] = oldv + self.alpha * (value - oldv)

    # Choose an action for the given state using an epsilon-greedy strategy
    def chooseAction(self, state):
        if random.random() < self.epsilon:  # With probability epsilon, choose a random action
            action = random.choice(self.actions)
        else:
            # Calculate Q-values for all possible actions in the current state
            q = [self.getQ(state, a) for a in self.actions]
            maxQ = max(q)  # Get the maximum Q-value
            count = q.count(maxQ)  # Count how many actions have this max Q-value
            if count > 1:  # If there's a tie, randomly choose one of the best actions
                best = [i for i in range(len(self.actions)) if q[i] == maxQ]
                i = random.choice(best)
            else:
                i = q.index(maxQ)  # Otherwise, select the action with the highest Q-value
            action = self.actions[i]
        return action

    # Learn method updates the Q-value for a (state, action) pair based on the reward and next state's Q-value
    def learn(self, state1, action1, reward, state2):
        # Calculate the maximum Q-value for the next state
        maxqnew = max([self.getQ(state2, a) for a in self.actions])
        # Use this max Q-value in the Q-learning update formula
        self.learnQ(state1, action1, reward, reward + self.gamma * maxqnew)

    # Print the Q-table with state-action pairs for debugging and analysis
    def printQ(self):
        keys = self.q.keys()
        states = list(set([a for a, b in keys]))  # Unique states
        actions = list(set([b for a, b in keys]))  # Unique actions
        dstates = ["".join([str(int(t)) for t in list(tup)]) for tup in states]
        print(" " * 4 + " ".join(["%8s" % ("(" + s + ")") for s in dstates]))
        for a in actions:
            print("%3d " % (a) + " ".join(["%8.2f" % (self.getQ(s, a)) for s in
states]))

    # Print the maximum Q-values for each state, representing the agent's
expected value function
    def printV(self):
        keys = self.q.keys()
        states = [a for a, b in keys]
        statesX = list(set([x for x, y in states]))  # Unique X-coordinates
        statesY = list(set([y for x, y in states]))  # Unique Y-coordinates
        print(" " * 4 + " ".join(["%4d" % (s) for s in statesX]))
        for y in statesY:
            maxQ = [max([self.getQ((x, y), a) for a in self.actions]) for x in
statesX]
            print("%3d " % (y) + " ".join([ff(q, 4) for q in maxQ]))  # Custom
formatting

# Helper function for formatting floating point numbers
import math
def ff(f, n):
    fs = "{:f}".format(f)
    if len(fs) < n:
        return ("{:" + str(n) + "s}").format(fs)  # Adjust to n-width if
shorter
    else:
        return fs[:n]  # Trim to n-width if too long
```

sarsa.py:

```
import random

# SARSA (State-Action-Reward-State-Action) agent class
```

```python
class Sarsa:
    # Initialize the SARSA agent with actions, epsilon, alpha, and gamma parameters
    def __init__(self, actions, epsilon=0.1, alpha=0.2, gamma=0.9):
        self.q = {}  # Dictionary to store Q-values for state-action pairs
        self.epsilon = epsilon  # Exploration rate for epsilon-greedy policy
        self.alpha = alpha  # Learning rate
        self.gamma = gamma  # Discount factor
        self.actions = actions  # List of possible actions

    # Method to retrieve the Q-value for a given (state, action) pair; defaults to 0.0 if it doesn't exist
    def getQ(self, state, action):
        return self.q.get((state, action), 0.0)

    # Update Q-value for (state, action) pair using the reward and expected future value
    def learnQ(self, state, action, reward, value):
        oldv = self.q.get((state, action), None)
        if oldv is None:  # If Q-value doesn't exist, initialize it with the reward
            self.q[(state, action)] = reward
        else:  # Otherwise, update Q-value using the SARSA update formula
            self.q[(state, action)] = oldv + self.alpha * (value - oldv)

    # Choose an action for the given state using an epsilon-greedy strategy
    def chooseAction(self, state):
        if random.random() < self.epsilon:  # With probability epsilon, choose a random action
            action = random.choice(self.actions)
        else:
            # Calculate Q-values for all possible actions in the current state
            q = [self.getQ(state, a) for a in self.actions]
            maxQ = max(q)  # Find the maximum Q-value
            count = q.count(maxQ)  # Count how many actions have this max Q-value
            if count > 1:  # If there's a tie, randomly choose one of the best actions
                best = [i for i in range(len(self.actions)) if q[i] == maxQ]
                i = random.choice(best)
            else:
                i = q.index(maxQ)  # Otherwise, select the action with the highest Q-value
            action = self.actions[i]
        return action
    # Learn method updates the Q-value for a (state, action) pair based on the
reward, next state, and next action
    def learn(self, state1, action1, reward, state2, action2):
        # Calculate the Q-value for the next state-action pair
        qnext = self.getQ(state2, action2)
        # Update Q-value for (state1, action1) using the SARSA update formula
        self.learnQ(state1, action1, reward, reward + self.gamma * qnext)
```