A5 Pole Balancing with Reinforcement Learning

For this assignment, you will write code for using reinforcement learning to learn to balance a pole. Follow the robot arm example in lecture notes `19 More Tic-Tac-Toe and a Simple Robot Arm`.

Download this implementation, cartpole_play.zip, of the pole-balancing problem. Unzip this file to get `cartpole_play.py`. This code requires the python packages `box2d` and `pygame`. You may install these using

```
conda install conda-forge::box2d-py
pip install pygame
```

After installing these packages and unzipping `cartpole_play.zip` you should be able to run

```
python cartpole_play
```

to see the cart pole animation. Push left and right on the cart with your keyboard arrow keys to try to balance the pole.

Define the class `CartPole` in a file named `cartpole.py`, following the `robot.py` example in notes 19. Copy the `QnetAgent` class from `robot.py` into your `cartpole.py` file and modify as necessary to call the necessary functions in `cartpole_play.py`. Define the `Experiment` class using the example in notes 19.

To define your `CartPole` class, you must define the critical environment functions from `rl_framework.py`. Try to design a reinforcement function that will lead to successful balancing. You should only need the pole angle, which is zero when the pole is balanced. Then your Qnet can be trained to minimize the sum of absolute values of the reinforcements. Or you could choose to define the reinforcement as -1 if the absolute value of the angle is greater than $0.75\,\pi$, 1 if less than $0.25\,\pi$ and zero otherwise.

To be clear and to help you get started, the structure of your `cartpole.py` file should look like

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import copy
from math import pi
import time
import pickle

import neuralnetworksA4 as nn
import rl_framework as rl  # for abstract classes rl.Environment and
rl.Agent

import cartpole_play as cp
```

```
class CartPole(rl.Environment):

    def __init__(self):
        self.cartpole = cp.CartPole()
        self.valid_action_values =  [-1, 0, 1]
        self.observation_size = 4  # x xdot a adot
        self.action_size = 1
        self.observation_means = [0, 0, 0, 0]
        self.observation_stds = [1, 1, 1, 1]  # not accurate but maybe
okay
        self.action_means = [0.0]
        self.action_stds =  [0.1]
        self.Q_means = [0.5 * pi]
        self.Q_stds = [2]

    def initialize(self):
        self.cartpole.cart.position[0] = np.random.uniform(-2., 2.)
        self.cartpole.cart.linearVelocity[0] = 0.0
        self.cartpole.pole.angle = 0  # hanging down
        self.cartpole.pole.angularVelocity = 0.0

    def reinforcement(self):

        state = self.observe()
        angle_magnitude = np.abs(state[2])

        if angle_magnitude > pi * 0.75:
            return -1
        elif angle_magnitude < pi * 0.25:
            return 1
        else:
            return 0

        # alternative:
        # return np.abs(angle)  # to be minimized

    # add other functions to your CartPole class as needed
    ...

######################################################################

class QnetAgent(rl.Agent):

    def initialize(self):
        env = self.env
        ni = env.observation_size + env.action_size
        self.Qnet = nn.NeuralNetwork(ni, self.n_hiddens_each_layer, 1)
        self.Qnet.X_means = np.array(env.observation_means +
env.action_means)
```

```
        self.Qnet.X_stds = np.array(env.observation_stds +
env.action_stds)
        self.Qnet.T_means = np.array(env.Q_means)
        self.Qnet.T_stds = np.array(env.Q_stds)

    # add other functions to your CartPole class as needed
    ...

######################################################################

class Experiment:

    def __init__(self, environment, agent):

        self.env = environment
        self.agent = agent

        self.env.initialize()
        self.agent.initialize()

    def train(self, parms, verbose=True):

        n_batches = parms['n_batches']
        n_steps_per_batch = parms['n_steps_per_batch']
        n_epochs = parms['n_epochs']
        method = parms['method']
        learning_rate = parms['learning_rate']
        final_epsilon = parms['final_epsilon']
        epsilon = parms['initial_epsilon']
        gamma = parms['gamma']

        ...

    # add other functions to your CartPole class as needed
    ...
```

Run your code as in this example:

```
import cartpole

cartpole_env = cartpole.CartPole()
agent = cartpole.QnetAgent(cartpole_env, [20, 20], 'max')

experiment = Experiment(cartpole_env, agent)

outcomes = experiment.train(parms)
```

with `parms` being parameters used by `Experiment`, such as

```
parms = {
    'n_batches': 2000,
    'n_steps_per_batch': 100,
    'n_epochs': 40,
    'method': 'scg',
    'learning_rate': 0.01,
    'initial_epsilon': 0.8,
    'final_epsilon': 0.1,
    'gamma': 1.0
}
```

The parameter values have not been chosen to best solve this problem. For the `verbose` output while training, print the mean of all reinforcements received so far.

To test performance of a trained agent, define a test function in your `Experiment` class like the following.

```
    def test(self, n_steps):

        states_actions = []
        sum_r = 0.0

        for initial_angle in [0, pi/2.0, -pi/2.0, pi]:

            self.env.cartpole.cart.position[0] = 0
            self.env.cartpole.cart.linearVelocity[0] = 0.0
            self.env.cartpole.pole.angle = initial_angle
            self.env.cartpole.pole.angularVelocity = 0.0


            for step in range(n_steps):

                obs = self.env.observe()
                action = agent.epsilon_greedy(epsilon=0.0)
                states_actions.append([*obs, action])
                self.env.act(action)
                r = self.env.reinforcement()
                sum_r += r

        return sum_r / (n_steps * 4), np.array(states_actions)
```

This function performs four runs, each one starting at a different `initial_angle`. Each experiment is run for `n_steps`. The function returns the mean of all reinforcement values over all four runs, and an array of all states and actions. You can run this function at the end of each training batch, collect the mean test reinforcements for each batch, and during `verbose` printing, include the mean of these test reinforcements so far. You may also use the mean reinforcement value to judge how well a particular set of parameter values work, printing a table like

```
           nh    nb   ns  ne  init epsilon  test r sum  exec minutes
177  [20, 20]  2000  200  40           0.8      0.1335      1.307003
64       [20]  2000  100   5           0.5      0.0650      0.245404
201  [40, 40]  2000  100  10           0.8      0.0295      0.448583
34       [10]  2000  200   5           0.5      0.0050      0.439274
76       [20]  2000  200   2           0.5     -0.0215      0.409806
...
```

I included execution times for each set of parameters just to see how long each training run took.

To see how well your agent is performing, plot some of the states returned by a final call to the `test` function. For example you can plot the angles for each step by

```
plt.plot(states_actions[:, 2])
```

Explain the design of your code, the experiments you ran, and how successful you were. Also describe any difficulties you ran in to.

There is no grading script for this assignment. You will be graded by the effort you put into running your experiments and the amount of detail you provide in your descriptions.

Check in a zip or tar file containing

- your A5 notebook
- `cartpole.py`
- `neuralnetworksA4.py`
- `optimizers.py`

Extra Credit

During training with various parameter values, save your best `Qnet` in a file using `pickle`. Once you have a saved a good agent, illustrate the performance of this agent by loading it from your `pickle` file and using it to control an animation of the cart-pole using the code in `cartpole_play.py` as a guide.

When you check in your A5 solution, include the pickle file containing your best `Qnet`. Your notebook must include code at the end for loading this file, running `test` with an agent using your `Qnet`, plotting the angle during the test runs, and animating the cart-pole being controlled by your agent with your best `Qnet`.