ML Assignment-2 (Goutham Deepak)

ABSTRACT

I have taken a code for a Reinforcement Learning algorithm from GitHub. It has been developed for the Tic-Tac-Toe game. The goal is to teach the two AI agents to play this game with each other. It uses Temporal Difference learning, where the AI learns from each move. Unlike Supervised learning, the agent doesn't know the correct move. Instead, it learns from its actions. The AI will improve its decision making skills by understanding the state of the game if it's a win, loss or a draw. There is a 3x3 board with 2 players who are marked as 1 and -1 and they make their move alternatively.

To track the state of the values an identifier called a hash is used. An epsilon-greedy strategy is applied. This means that random moves with a low probability are explored so that new strategies can be tried out. But most of the time, it moves based on the highest estimated value. This will help the AI discover new strategies while considering what it has learned. There are different classes and functions in the code. The State class represents the board and tells us the state of the game. The reinforcement learning logic is in the Player class. This helps in the estimation and for also making the moving decisions. This is also where the training happens. As training progresses, the agents learn from the outcomes by adjusting the value function. The Judger class helps to alter the players and also to reset the board for each new game.

So the code I have taken is a very structured code used to train AI players to play Tic-Tac-Toe optimally. I believe that if trained properly, it will at least secure a draw against a human or another AI bot. This is an example of how AI can learn and adapted to maximize rewards in a game environment and can improve by using continuous feedback.

I have also added the comments for the 2 core functions which for reinforced learning. They are the act( ) and the step( ) function.

Functions

act ( ) function:

It is responsible for deciding which action has to be taken next. First it chooses a random action which happens with a probability of epsilon. This will help the bandit go through new actions. If the Upper Confidence Bound is there then it will calculate it and pick the highest. If gradient based approach is used, then it will convert the values into probabilities. So the one with more estimated reward will be chosen.

step() Function:

It handles what happens after an action is taken. The reward is calculated. It has some randomness, and it will update the total count for the action. It will update the estimated values using sampling averages, gradient based update or take a constant step size. It is mainly useful as the bandit will learn from the actions and improve over time based on the rewards it has received.

```python
##########################################################################
# Copyright (C)                                                          #
# 2016-2018 Shangtong Zhang(zhangshangtong.cpp@gmail.com)                #
# 2016 Tian Jun(tianjun.cpp@gmail.com)                                   #
# 2016 Artem Oboturov(oboturov@gmail.com)                                #
# 2016 Kenta Shimada(hyperkentakun@gmail.com)                            #
# Permission given to modify the code as long as you keep this           #
# declaration at the top                                                 #
##########################################################################

import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from tqdm import trange

matplotlib.use('Agg')


class Bandit:
    # This is for initializing the Bandit class with various parameters.
    def __init__(self, k_arm=10, epsilon=0., initial=0., step_size=0.1, sample_averages=False,
                 UCB_param=None, gradient=False, gradient_baseline=False, true_reward=0.):
        self.k = k_arm
        self.step_size = step_size
        self.sample_averages = sample_averages
        self.indices = np.arange(self.k)
        self.time = 0
        self.UCB_param = UCB_param
        self.gradient = gradient
        self.gradient_baseline = gradient_baseline
        self.average_reward = 0
        self.true_reward = true_reward
        self.epsilon = epsilon
        self.initial = initial


    def reset(self): # The bandit is reset before each run.
        self.q_true = np.random.randn(self.k) + self.true_reward
        self.q_estimation = np.zeros(self.k) + self.initial
        self.action_count = np.zeros(self.k)
        self.best_action = np.argmax(self.q_true)
        self.time = 0

    # This function helps to select the next action for the bandit.
    def act(self):
        # It first checks if a random number is less than epsilon. If it is then the bandit will shoose a random number and explore it
```


```python
    # This function helps to select the next action for the bandit.
    def act(self):
        # It first checks if a random number is less than epsilon. If it is then the bandit will shoose a random number and explore it
```


```python
        if np.random.rand() < self.epsilon:
            return np.random.choice(self.indices)

        if self.UCB_param is not None: # If Upper Confidence Bound is being used then it calculates the UCB values.
            UCB_estimation = self.q_estimation + \
                self.UCB_param * np.sqrt(np.log(self.time + 1) / (self.action_count + 1e-5))  # Confidence bound is added to the estimates
            q_best = np.max(UCB_estimation) # Action with the highest UCB value is found
            return np.random.choice(np.where(UCB_estimation == q_best)[0]) # If many have the same value then one is picked at random

        if self.gradient: # If the gradient method is used then it converts the estimates into probabilities
            exp_est = np.exp(self.q_estimation) # The exponent is taken to make the probabilities
            self.action_prob = exp_est / np.sum(exp_est) # It values are nromalized to make the probabilities
            return np.random.choice(self.indices, p=self.action_prob) # It choose the action

        q_best = np.max(self.q_estimation) # If epsilon-greedy or UCB is not used then it will pick the action with the highest estimated value
        return np.random.choice(np.where(self.q_estimation == q_best)[0]) # If there are multiple actions with the same value, it picks one randomly.

    # This function helps to perform the action and update the estimates.
    def step(self, action):
        reward = np.random.randn() + self.q_true[action] # It calculates the reward for the action and adds some radomness
        self.time += 1 # It increases the time step by 1
        self.action_count[action] += 1 # It increases the count of how many times the action was taken
        self.average_reward += (reward - self.average_reward) / self.time # It updates the average reward

        if self.sample_averages: # If using sample averages
            self.q_estimation[action] += (reward - self.q_estimation[action]) / self.action_count[action] #it updates the estimate for the action


        elif self.gradient: # Or else if gradient method is used the gradients is used for updation
            one_hot = np.zeros(self.k)  # It creates a one_hot vector for the action
            one_hot[action] = 1  # It sets the position of the chosen action to 1


            if self.gradient_baseline: # It uses the average reward as a baseline
                baseline = self.average_reward
            else:
                baseline = 0  # Otherwise it keeps the baseline to 0

            self.q_estimation += self.step_size * (reward - baseline) * (one_hot - self.action_prob) # Gradient formula is used to update the estimates

        else:
            self.q_estimation[action] += self.step_size * (reward - self.q_estimation[action]) # Or else constant step size is used for the updation

```

Untitled-1

```python
    # This function helps to perform the action and update the estimates.
    def step(self, action):
        reward = np.random.randn() + self.q_true[action] # It calculates the reward for the action and adds some radomness
        self.time += 1 # It increases the time step by 1
        self.action_count[action] += 1 # It increases the count of how many times the action was taken
        self.average_reward += (reward - self.average_reward) / self.time # It updates the average reward

        if self.sample_averages: # If using sample averages
            self.q_estimation[action] += (reward - self.q_estimation[action]) / self.action_count[action] #it updates the estimate for the action


        elif self.gradient: # Or else if gradient method is used the gradients is used for updation
            one_hot = np.zeros(self.k)  # It creates a one_hot vector for the action
            one_hot[action] = 1  # It sets the position of the chosen action to 1


            if self.gradient_baseline: # It uses the average reward as a baseline
                baseline = self.average_reward
            else:
                baseline = 0  # Otherwise it keeps the baseline to 0

            self.q_estimation += self.step_size * (reward - baseline) * (one_hot - self.action_prob) # Gradient formula is used to update the estimates

        else:
            self.q_estimation[action] += self.step_size * (reward - self.q_estimation[action]) # Or else constant step size is used for the updation

        return reward # The reward for the action is returned


def simulate(runs, time, bandits):
    rewards = np.zeros((len(bandits), runs, time))
    best_action_counts = np.zeros(rewards.shape)
    for i, bandit in enumerate(bandits):
        for r in trange(runs):
            bandit.reset()
            for t in range(time):
                action = bandit.act()
                reward = bandit.step(action)
                rewards[i, r, t] = reward
                if action == bandit.best_action:
                    best_action_counts[i, r, t] = 1
    mean_best_action_counts = best_action_counts.mean(axis=1)
    mean_rewards = rewards.mean(axis=1)
    return mean_best_action_counts, mean_rewards

```

Ln 1, Col 72 Spaces: 4 UTF-8 LF Python 3.11.5 ('base': conda) Go Live