Name: Kush Harish Vora
Course: CSE584
Homework-2
22nd October 2024

Q-Learning Algorithm for Travelling Salesman Problem

Abstract:

The provided code [1] showcases an application of reinforcement learning (RL) to address the Travelling Salesman Problem (TSP) [2]. TSP is a classic challenge in the field of combinatorial optimization. The problem involves finding the shortest possible route that visits all the cities in the dataset exactly once and then returns back to the start point. TSP has numerous real-world applications in fields such as logistics, transportation, and even in hardware— circuit designs. Traditional solutions to TSP rely on heuristic algorithms, which suffer from limited scalability or adaptability in dynamic environments. The provided code leverages Q-learning [3], which is a model-free RL algorithm, that iteratively optimizes the travel sequence (route). Here, each city is modeled as a state and all the travel decisions as actions. Here Q-tables are constructed which are used to evaluate the quality of each state-action pair. Two types of rewards are incorporated viz. immediate (inverse of the euclidean distance, hence giving more reward to shorter routes) and delayed rewards which are used to capture the long-term benefit of visiting particular (pivotal) cities. An epsilon-greedy exploration strategy is incorporated during the training process. This is done to balance exploration of unvisited routes and exploitation of learned (optimal) actions. In conclusion, the code shows how RL can be applied to solve TSP in static as well as dynamic environments.

Code: (https://github.com/kushv16/CSE584-HW2/blob/master/rl.py)

```
import psycopg2
import numpy as np
import random

# Connecting to the database which holds the data
connection = psycopg2.connect(user='usname', database='db_name')
cursor = connection.cursor()

# Execute a SQL query to retrieve city locations (x, y coordinates) from the 'city_locations' table
cursor.execute("select x_coor, y_coor from city_locations;")
# Fetch all the results (city coordinates) and store them into a 'locations'
locations = cursor.fetchall()
```

```
# Number of destinations (cities) we have to visit
n_dest = len(locations)

# Initialize a distance matrix to store the distances between every pair of cities
# Create a matrix filled with zeros of size [n_dest, n_dest]
dist_mat = np.zeros([n_dest, n_dest])

# Loop over each pair of cities to calculate and store the Euclidean distance between them
for i, (x1, y1) in enumerate(locations):
    for j, (x2, y2) in enumerate(locations):
        # Calculate the Euclidean distance between city (x1, y1) and city (x2, y2)
        d = np.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)
        dist_mat[i, j] = d  # Store the calculated distance in the distance matrix


"""
This function updates the Q-table using the Q-learning formula.

Q-tables are lookup tables which store the cumulative reward obtained by taking
a given action in a given state and following the optimal policy afterward.

The function takes the following input parameters
1. q: Q-table
2. dist_mat: Matrix which contains the distance between all the cities
3. state: Current city, which is state in the context of reinforcement learning
4. action: Next city to visit, which is the action in context of reinforcement learning
5. alpha: Learning rate which controls how much new information should influence the Q-value
6. gamma: Discount factor which determines how much we value future rewards
"""
def update_q(q, dist_mat, state, action, alpha=0.012, gamma=0.4):
    """
        We start by calculating the immediate reward.
        Our goal in the travelling salesman problem should be to minimize the distance to complete the
task.
        Hence, we use the inverse of the distance to give higher reward to cities that are closer
    """
    immed_reward = 1. / dist_mat[state, action]

    """
        Next, we calculate the delayed reward.
        We look at the best possible future state, in order to maximize future rewards
    """
    delayed_reward = q[action, :].max()
    # Update the Q-value table for the current state-action pair using the Q-learning formula
    q[state, action] += alpha * (immed_reward + gamma * delayed_reward - q[state, action])

    # Return the updated Q-table
    return q

# Initialize the Q-table: it's a matrix of size [n_dest, n_dest] where all values start at zero
q = np.zeros([n_dest, n_dest])


"""
    Parameters for training
    1. Epsilon is the exploration rate, which controls how often the agent explores new actions vs.
exploiting known ones
    2. Number of training epochs or iterations where the agent tries to learn
"""
epsilon = 1.
n_train = 2000

# Begin training
for i in range(n_train):
    # Start the trip from the first city (city 0)
    traj = [0]
     # The initial state is the first city (city 0)
    state = 0

    # Create a list of possible actions (cities to visit that have not been visited yet)
    # This needs to be maintained, because we cant visit the same city twice else it would just be an
infinite cyclic loop
    possible_actions = [dest for dest in range(n_dest) if dest not in traj]

    # We cant visit a city more than once.
    while possible_actions:
        # Use an epsilon-greedy strategy to decide between exploring new actions or exploiting known
actions
        if random.random() < epsilon:  # Exploration: with probability epsilon, choose a random action
             # Randomly choose the next city (action)
            action = random.choice(possible_actions)
        else:  # Exploitation: with probability 1 - epsilon, choose the action with the best Q-value
            # Find the best action based on the current Q-table
            best_action_index = q[state, possible_actions].argmax()
             # Choose the action with the highest Q-value
            action = possible_actions[best_action_index]
        # Update the Q-table with the newly chosen action
        q = update_q(q, dist_mat, state, action)

        # Add the chosen action (next city) to the sequence of cities visited
        traj.append(action)

        # Update the state to the new city (the most recent city added to the trajectory)
        state = traj[-1]

        # Update the list of possible actions (remove the visited city from the list)
        possible_actions = [dest for dest in range(n_dest) if dest not in traj]

    # After visiting all cities, return to the first city (city 0)
    action = 0
    # Update the Q-table for the return trip
    q = update_q(q, dist_mat, state, action)
    # Add the first city to the end of the trajectory
    traj.append(0)

    # We decay the exploration rate (epsilon) after each iteration to gradually shift from exploration to
exploitation
    epsilon = 1. - i * 1/n_train


# Inference phase (find the best route after training)
traj = [0]  # Start from the first city
state = 0  # Initial state is the first city
distance_travel = 0.  # Initialize total distance traveled

# Find the optimal path based on the trained Q-table
possible_actions = [dest for dest in range(n_dest) if dest not in traj]  # List of unvisited cities
while possible_actions:
    # Select the best action (city) using the learned Q-values
    best_action_index = q[state, possible_actions].argmax()  # Find the best action based on Q-table
    action = possible_actions[best_action_index]  # Choose the next city

    # Add the distance of the chosen action to the total distance traveled
    distance_travel += dist_mat[state, action]

    # Add the chosen city to the trajectory
    traj.append(action)

    # Update the state to the new city
    state = traj[-1]
    # Update the list of possible actions (remove the visited city from the list)
    possible_actions = [dest for dest in range(n_dest) if dest not in traj]

# After visiting all cities, return to the first city (city 0)
action = 0  # The action to return to the first city
distance_travel += dist_mat[state, action]  # Add the return trip distance to the total
traj.append(0)  # Add the warehouse to the trajectory

# Print the best trajectory found and the total distance traveled
print('Best trajectory found:')
print(' -> '.join([str(b) for b in traj]))  # Print the sequence of cities visited
print(f'Distance Travelled: {distance_travel}')  # Print the total distance traveled
```

Sources:

1. https://www.cybertec-postgresql.com/en/implementation-of-a-reinforcement-learning-algorithm-from-scratch/
2. Gavish, Bezalel, and Stephen C. Graves. "The travelling salesman problem and related problems." (1978).
3. Watkins, Christopher JCH, and Peter Dayan. "Q-learning." *Machine learning* 8 (1992): 279-292.