Demonstrate that a Distributed Supply Chain Problem can be Managed by Co-Operating AI Agents

Component 1

1 Task

Create a supply chain environment to train AI agents that play the supply chain management (game).

2 Team

THE_OUTLIERS:
Apoorva Vikram Singh
Divyansha
Anubhav Sachan
Asis Kumar Roy
Yash Srivastava

College: National Institute of Technology Silchar

3 Introduction

Supply-chain optimization is the application of various processes to guarantee the optimal operation of a manufacturing and distribution supply chain. It is a problem faced by many companies with a factory and multiple warehouses. A crucial decision that needs to be made is determination of amount of products that needs to be produced by that factory and the stock that should be build up in the warehouses. This supply chain management can be done manually by small companies but automation becomes necessary when a large scale company comes into play. This automation can be achieved by using reinforcement learning techniques. Reinforcement Learning focuses on learning to make a good set of sequential decisions in the model of a world. An agent interacts with the environment (model of the world) to record its experiences based on the set of actions taken. The experiences act as a base for the agent to learn efficiently. The primary goal in learning is to extend the number of good experiences in the shortest time possible.

4 Environment Setup

The environment consists of one factory and multiple warehouses. The warehouses are of two types. One central factory warehouse and W distribution warehouses are modeled.

4.1 Defining the Variables

Let's assume that a factory's production level at a time t is $b_{0,t}$ with a constant cost of c_0 rupees per unit.
The maximum storage capacity of the factory warehouse is m_0 units with c_0^S storage cost per unit for one time step with l_0^t stock level at time t.
At any time t $b_{j,t}$ units, will be shipped from factory warehouse to distribution warehouse j with a transportation cost of c_j^T rupees per unit.
The distribution warehouse j has a storage cost of c_j^S, a maximum capacity of m_j and a stock level l_j^t at time t.
Demand from a distribution warehouse at time t is $d_{j,t}$ units. The selling price of the product sold is p. Ideally the demand should never exceed the stock level at a distribution warehouse thus a penalty of c_j^P per unfulfilled unit is imposed. These unfulfilled demands are carried over time steps which are modeled as negative stock level.

4.2 Reward Function

The reward function for this environment consist of 5 parts. These terms are defined for each time step:

- Revenue -: $p \sum_{j=0}^{W} d_j$
- Production Cost -: $c_0 b_0$
- Total Storage Cost -: $\sum_{j=0}^{W} c_j^S max\, l_j, 0$
- Transportation Cost -: $\sum_{j=1}^{W} c_j^T b_j$
- Penalty Cost -: $\sum_{j=1}^{W} c_j^P min\, l_j, 0$

Thus we get the equation of our reward function:

$$r = p \sum_{j=0}^{W} d_j - c_0 b_0 - \sum_{j=0}^{W} c_j^S max\, l_j, 0 - \sum_{j=1}^{W} c_j^T b_j + \sum_{j=1}^{W} c_j^P min\, l_j, 0$$

4.3 States and Actions

The state vector includes demand values and current stock levels for all warehouses for several previous steps:

$$s_t = (l_{0,t}, l_{1,t}, \dots, l_{W,t}, d_{t-1}, \dots, d_{t-\tau})$$

where,

$$d_t = (d_{1,t}, \dots, d_{W,t})$$

The state update rule is defined as -:

$$\begin{aligned} s_{t+1} = \big(&min[l_{0,t} + b_0 - \sum_{j=1}^{W} b_j, m_0], \quad (Factory\ stock\ update) \\ &min[l_{1,t} + b_{1,t} - d_{1,t}, m_1], \quad (Warehouse\ stock\ update) \\ &\dots, \\ &min[l_{W,t} + b_{W,t} - d_{W,t}, m_W], \\ &d_t, \\ &\dots, \\ &d_{t-\tau}\big) \end{aligned}$$

Finally the action vector can be represented by -:

$$a_t = (b_{0,t}, \dots, b_{W,t})$$

4.4 Simulation of demand function

The demand is simulated using the demand function given below. It incorporates seasonal demand changes and a stochastic component.

$$d_{j,t} = \frac{d_{max}}{2}\left(1 + sin\left(\frac{d_{4\pi(t+2j)}}{T}\right)\right) + \mu_{j,t}$$

Here μ is a random variable of uniform distribution

5 Implementation details

We have used Pytorch 1.6 for implementation of the above details. The implementation of each component of the environment is described below. The repository is available on Github.

5.1 State and action

The code used for implementing the state and action as described in section 4.3 in python:

```python
class Action(object):
    """
    shipping to warehouses and changes in production level
    """

    def __init__(self, no_of_warehouses):
        self.production_level = 0  # b(0,t)
        self.shippings = np.repeat(0, no_of_warehouses)  # W(j,t)


class State(object):
    """
    defining the state of the environment
    """
    def __init__(self, no_of_warehouses, demand_history, eps, t=0):
        self.no_of_warehouses = no_of_warehouses  # W
        self.demand_history = demand_history
        self.factory_stock = 0  # l(0,t)
        self.warehouse_stock = np.repeat(0, no_of_warehouses)  # l(
    j,t)
        self.eps = eps
        self.t = t

    def array_state(self):
        return np.concatenate(([self.factory_stock], self.
    warehouse_stock, np.hstack(self.demand_history), [self.t]))

    def stock_only(self):
        return np.concatenate(([self.factory_stock], self.
    warehouse_stock))
```

Listing 1: state action implementation in python

5.2 Demand Function

The code used for simulating a demand function and sampling as described in section 4.4 in python is given below:

```python
def demand(self, t, j):
"""
:param t: time at which demand is needed
:param j: warehouse for which demand is needed
:return: returns demand at time t for warehouse j
"""

return np.round(self.d_max / 2 + self.d_max / 2 * np.sin(2 * np.pi
    * (t + 2 * j) / self.eps * 2) + np.random.randint(0, self.d_var)
    )
```

Listing 2: Demand function implementation in python

5.3 Taking a step

The code used for taking a step i.e. transitioning to the next state and subsequently calculating rewards (total profit) for reaching this particular state as described in section 4.2 in python is given by Listing 3:

```python
def step(self, state, action):
    """
    :param state: state instance
    :param action: action instance
    :return: next_state,rewards,completed episodes or not
    """

    demands = np.fromfunction(lambda j: self.demand(self.t, j + 1), (self.no_of_warehouses,), dtype=int)

    # For Rewards
    total_revenue = self.unit_price * np.sum(demands)
   production_cost = self.unit_cost * action.production_level

    total_storage_cost = np.dot(self.storage_costs,      np.maximum(state.stock_only(), np.zeros(self.no_of_warehouses + 1)))

    transportation_cost = np.dot(self.transporation_costs, action.shippings)

    penalty_cost = -1 * self.penalty_unit_cost * ( np.sum(np.minimum(state.warehouse_stock, np.zeros(self.no_of_warehouses))) + min(state.factory_stock, 0))

    rewards = total_revenue - production_cost - total_storage_cost - penalty_cost - transportation_cost

    # For Next state
    next_state = State(no_of_warehouses=self.no_of_warehouses, demand_history=list(self.demand_history), eps=self.eps, t=self.t)

    next_state.factory_stock = min(state.factory_stock + action.production_level - np.sum(action.shippings), self.storage_capacities[0])

    for i in range(self.no_of_warehouses):
        next_state.warehouse_stock[i] = min( state.warehouse_stock[i] + action.shippings_to_warehouses[i] - demands[i],   self.storage_capacities[i + 1])

    self.t += 1
    self.demand_history.append(demands)

    return rewards, next_state, self.t == self.eps - 1
```

Listing 3: Reward Function and Next State implementation in python

6 Conclusion and future work

The environment setup has been done in order to simulate a supply chain optimization problem. We are planning to use Deep Deterministic Policy Gradients (DDPG) Algorithm to optimize our problem statement due to the fact that DDPG algorithm combines the Actor-Critic paradigm with the stabilization techniques from Deep Q-Learning. Using DDPG gives us multiple options for the optimization such as inclusion of soft updates for the critic network.