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

Component 2

1 Task

Create algorithm agents that follow a logic to manage item locations. The repository for all the code is available on Github

2 Team

THE_OUTLIERS:

Apoorva Vikram Singh
Divyansha
Anubhav Sachan
Asis Kumar Roy
Yash Srivastava

College: National Institute of Technology Silchar

3 Recapitulation

In the previous task, we defined reward function, environment, states, actions, transition function and demand function.

4 Baseline

Having defined the environment before, a baseline needs to be established for the supply chain control policy. The $(s,\ Q)$ Policy is one of the conventional solutions. This policy states that order your Economic Order Quantity Q, every time your inventory position drops below s (Reorder Point). In simple terms, compare the stock level with reorder point s at every time step. If the stock level falls under the reorder point, reorder Q units. Otherwise, do not take any actions. A saw-tooth stock level pattern of the (s, Q) Policy can be seen in the fig. 1

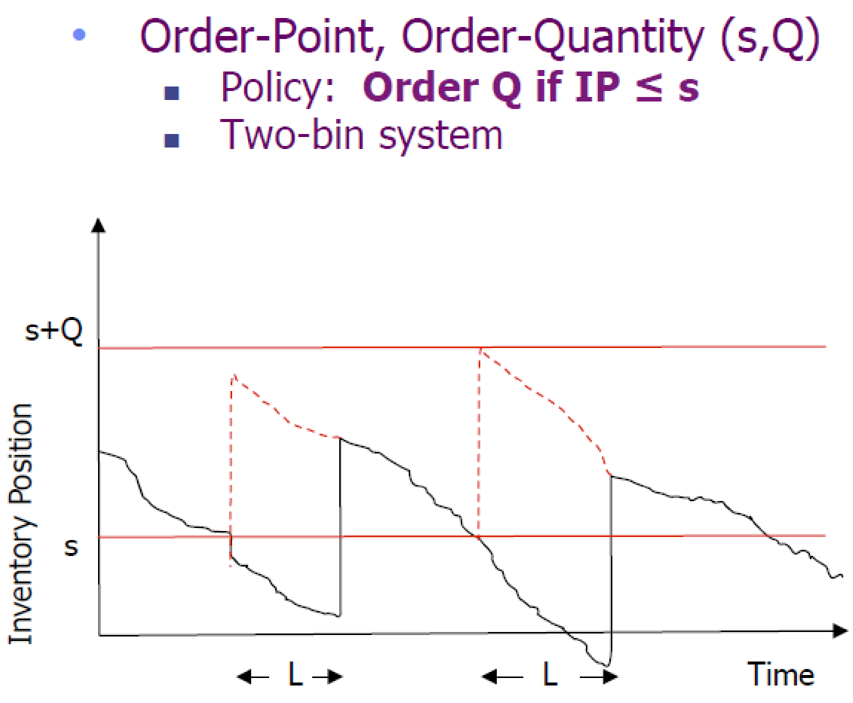

Figure 1: (s,Q) policy for inventory management

For each warehouse, reordering decisions are made independently of other warehouses. For different warehouses, the policy parameters s and Q can vary.

5 Simulation of (s, Q) Policy

The simulation of policy is in PyTorch and the code snippet is given below.:

```python
class BaselinePolicy(object):
    """
    defining the baseline policy for the environment
    """

    def __init__(self, st_fac_safe, qty_fac, st_war_saf, qty_war):
        """
        :param st_fac_safe: safety stock in the factory
        :param qty_fac: qty to be reordered (from production
  building) for items are out of stock in factory
        :param st_war_saf: safety stock in warehouse
        :param qty_war: amount to be reordered from factory.
        """
        self.st_fac_safe = st_fac_safe
        self.qty_fac = qty_fac
        self.st_war_saf = st_war_saf
        self.qty_war = qty_war

    def choose_action(self, state):
        """
        :param state: state instance defining the current state we are at.
        :return:action performed
        """
        action = Action(state.no_of_warehouses)
        for i in range(state.no_of_warehouses):
            if state.warehouse_stock[i] < self.st_war_saf[i]:
                action.shippings[i] = self.qty_war[i]

        if state.factory_stock - np.sum(action.shippings) < self.st_fac_safe:
            action.production_level = self.qty_fac
        else:
            action.production_level = 0
        return action


def eps_sim(envt, policy):
    state = envt.initial_state()
    transitions = []

    for _ in range(envt.eps):
        action = policy.choose_action(state)
        state, rewards, _ = envt.step(state, action)
        transitions.append([state, action, rewards])

    return transitions


def sim(envt, policy, num_episodes):
    r_array = []
    for episode in range(num_episodes):
        envt.reset()
        r_array.append(sum(np.array(eps_sim(envt, policy)).T[2]))
    return r_array
```

Listing 1: Baseline Policy and its simulation function as defined in main.py.

6 Optimization of Baseline Policy

Since we have 10 parameters in our environment (5 (s, Q) pairs), setting policy parameters is a challenging task. The parameters need to be selected in such a way that shortage costs and storage costs are balanced in the case of uncertain demands. To be more specific, the reorder points needs to have the ability of cushion demand shocks to some extent. If the demand distribution parameters are known, analytical approach can be taken. We have, instead, tried to solve this problem using brute force search. We use Adaptive Experimentation Platform released by Facebook to conduct the brute force search through the parameter space. Adaptive Experimentation Platform utilizes Bayesian optimization and provides a convenient API to perform experimentation. The snippets of code for optimization of parameters for the (s, Q) Policy has been given below:

```python
def f_policy(p):
    policy = BaselinePolicy(
        p['factory_s'],
        p['factory_Q'],
        [p['w1_s'], p['w2_s'], p['w3_s'], p['w4_s']],
        [p['w1_Q'], p['w2_Q'], p['w3_Q'], p['w4_Q']]
    )
    return np.mean(sim(envt, policy, num_episodes=30))

best_parameters, best_values, experiment, model = optimize(
    parameters=[
            {"name": "factory_s", "type": "range", "bounds": [0.0, 30.0], },
            {"name": "factory_Q", "type": "range", "bounds": [0.0, 30.0], },
            {"name": "w1_s", "type": "range", "bounds": [0.0, 20.0], },
            {"name": "w1_Q", "type": "range", "bounds": [0.0, 20.0], },
            {"name": "w2_s", "type": "range", "bounds": [0.0, 20.0], },
            {"name": "w2_Q", "type": "range", "bounds": [0.0, 20.0], },
            {"name": "w3_s", "type": "range", "bounds": [0.0, 20.0], },
            {"name": "w3_Q", "type": "range", "bounds": [0.0, 20.0], },
            {"name": "w4_s", "type": "range", "bounds": [0.0, 20.0], },
            {"name": "w4_Q", "type": "range", "bounds": [0.0, 20.0], },
        ],
    evaluation_function=f_policy,
    minimize=False,
    total_trials=200,
    # 200 is the smallest, yielding good results, can be extended for more trials as well.
    )
```

Listing 2: Optimization using Facebook Ax (Source: main.py).

This optimization is then combined with grid search fine tuning to obtain a profit of 4758. More insight into the policy behavior can be gained by the visualization of how the shipments, profits, stock levels, and production levels change over time. This can be seen in figure 2.

The random component of the demand is relatively small, and it makes more sense to ship products on an as-needed basis rather than accumulate large safety stocks in distribution warehouses. This is clearly visible in the above plots: the shipment patterns loosely follow the oscillating demand pattern, while stock levels do not develop a pronounced saw-tooth pattern.

7 Conclusion and future work

The environment setup has been done in order to simulate a supply chain optimization problem. We have simulated a baseline policy in order to achieve an cumulative

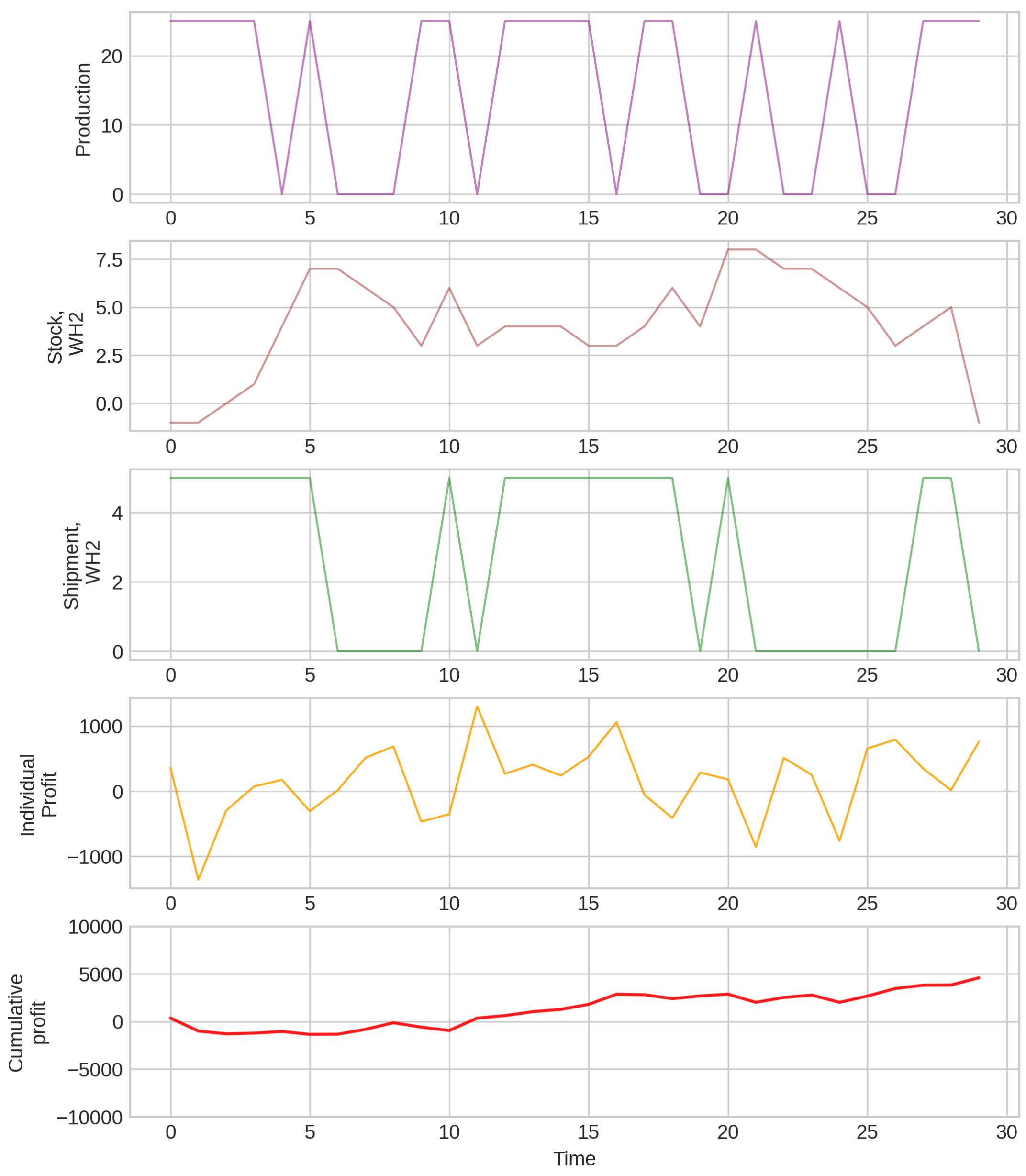

Figure 2: Simulation results

profit of approximately Rs. 4758. From figure 2, we can observe that the shipment pattern of warehouse 2 is oscillating in nature. This is due to a relatively small variable component in our demand and therefore it is more sensible to ship products on an as-needed basis instead of hoarding huge safety stocks in warehouses.

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.