
**MASTER'S DEGREE COURSE IN MANAGEMENT ENGINEERING**

## LABORATORY PROJECT 2 – Q learning

## NETWORK RESOURCE MANAGEMENT

# Product Assembly and Inventory Control

**Members**: Francesco Sanetti 1933718

Alessandro Rossi 1941142

Alberto Magro 1935939

**Professor:** Paolo Di Lorenzo

**Academic Year 2023/24 - II semester**

## Summary

## 1 - Introduction

We are considering a manufacturing plant that purchases lots of raw material to assembly and sell products to customers. The aim of the algorithm is to identify the optimal policy that maximises the profit using Q learning.

**Costs:**

-Unit purchasing cost

-Unit backorder cost

-Unit holding cost

-Fixed order cost

**Inventory:**

-Materials

-Products

**Assumptions:**

Products get assembled and sold in the same time slot in which they are requested

Purchased materials will arrive in the successive time slot

In order to teach the agent, if an inventory is larger than its maximum the reward is updated negatively, as we are ordering something that will be discarded

## Demand

The demand is composed by 3 elements:

-Base demand

-Price sensitivity

This component obviously represents a direct relation with the price choice action

-Previous sells sensitivity (up to N days before)

The coefficient related to this component exponentially decreases overtime

Even if demand levels are not present in the Q table, we defined multiple relations with actions' elements so that the agent's behaviour indirectly affects the demand

Backorder

-If the demand cannot be satisfied, the firm will incur in backorders, representing an additional cost that reduces the reward (defined as the profit of the firm).

-If possible, backorders are satisfied at the first available time slot before looking at the current demand.

-In order to satisfy backorders the firm will first use the products inventory, then (if needed) they will assemble other products using materials from the inventory.

-The firm can cumulate up to a maximum level of backorders.

## 2 – Single product

Consider a machine that inputs m raw materials and outputs k finished products: each product k is assembled according to a well-defined recipe of m raw materials. We aim to use a reinforcement learning algorithm to find the optimal policy for ordering the best quantity of raw materials and setting the selling price for the finished products for every possible state. At each time slot, the model receives information about the state (inventory of raw materials, inventory of finished products, and the amount of backorder to be fulfilled) and makes necessary decisions to meet customer orders, minimize costs (ordering, holding, and backorder costs), and maximize profits from selling products.

**Assumptions**

- If raw materials required by the recipe are available, the products are assembled and sold in the same time slot $t$
- Purchased raw materials will arrive in the time slot $t + 1$.
- Demand is always met (if possible).
- If demand is not met, a backorder cost is incurred, and the orders must be fulfilled in the upcoming time slots.
- Backorders are fulfilled before current demand.
- Backorders must be satisfied at the first available time slot (if possible).

**The State**

- $IMG\_m$ is the quantity available of raw material $m$.
- $IPF\_k$ is the inventory level of product $k$.
- $B\_k$ is the backorder for product $k$.

To simulate real-world uncertainties, a small random variation around $IMG\_m$ is introduced. In this way we have a random transition probability between the states.

**Actions**

In each time slot, the actions taken are:

1. how much raw material $G$ to buy for each type of material $m$.
2. set the price $P$ for each assembled product $k$.
3. How much product Q the plant will produce

The set of possible action $\mathcal{A}$ is given by all possible combinations of $G_m$, $P_k$, $Q_k$:

$$\mathcal{A} = \{(G_m, P_k, Q_k) \mid G_m \in \mathcal{G}_m, P_k \in \mathcal{P}_k, Q_k \in \mathcal{Q}_k, m = 1, \dots, M, k = 1, \dots, K\}$$

Each element $(G\_m, P\_k, Q\_k)$ represents a specific action.
Since Q-learning assumes that the agent makes one decision at a time, individual actions are treated as a single combined action.

**Demand**

The current demand $D_k(t)$ for a product k depends not only on the current price $P_k(t)$ but also on the quantity sold $V_k(t-1)$ in the previous time slot.
If you sell a lot in the current slot, future demand may decrease.

$$D_k(t) = f_k\big(P_k(t)\big) - \eta \cdot V_k(t-1) + \epsilon_k(t)$$

Where:

- $f_k\big(P_k(t)\big) = a_k \_ b_k \cdot P_k(t)$ is the linear demand function based on price.
- $V_k(t-1)$ is the quantity sold of product k in the previous time slot.
- $\eta$ is a positive coefficient capturing the negative effect of the previous sale on current demand.
- $\epsilon_k(t)$ is the random noise capturing uncertainty in demand.

This model implies that if the agent sets a low price in each time slot (leading to high sales), the demand might decrease in the next time slot, encouraging the agent to consider this trade-off in their pricing and production strategy.

**Extended Demand Model**

To capture long-term effects (e.g., high sales affecting not just the next slot's demand but also subsequent ones), we have extended the demand model to include a feedback term that considers past time slots:

$$D_k(t) = f_k\big(P_k(t)\big) - \sum_{i=1}^{N} \eta_i \cdot V_k(t-i) + \epsilon_k(t)$$

Where $\eta_i$ are coefficients weighting the effect of sales in the previous N time slots.

**Markov Property**

In our model, we chose not to include demand in the state as it would have violated the Markov property: the agent's actions would affect not only the next state but also future states. Demand is used only to calculate the reward based on the chosen price and previous sales, ensuring that the Markov property is not violated and convergence to an optimal policy is guaranteed.

Moreover, not including demand in the state was also a modeling choice: in reality, we rarely know with certainty what the demand we need to meet is. Thus, we hypothesized that the agent cannot observe demand.

**Backorders**

Unmet demand in each time slot will be fulfilled in the future, but with an additional backorder cost.

Let $B_k(t)$be a variable representing for each product k the quantity of unmet demand in the time slot t:

$$B_k(t+1) = \max[0, B_k(t) + D_k(t) - V_k(t)]$$

Therefore, the total demand that needs to be fulfilled in slot t+1 becomes:

$$D'_k(t+1) = D_k(t+1) + B_k(t+1)$$

**Reward**

The reward depends on:

- Profit: $\pi(t) = \sum_{k=1}^{K} \mathrm{V}_k(t) \cdot P_k(t)$
- Total holding cost: $H(t) = C_H \cdot (\sum_{m=1}^{M} \quad IMG_m + \sum_{k=1}^{K} IPF_k)$

Where $C_H$ represents a variable cost that follows a normal distribution.

- Total ordering cost: $O(t) = \sum_{m=1}^{M} (G_m \cdot \mathrm{Cmat}_m)$

The backorder cost is modeled as proportional to the total backorder level.

- Backorder Cost: $\mathrm{C}_{\mathrm{bk}}(t) = \sum_{k=1}^{K} Cb \cdot B_k(t)$

Thus, the final reward will be:

$$\mathrm{R}(t) = \pi(t) - H(t) - O(t) - \mathrm{C}_{\mathrm{bk}}(t)$$

**Two different codes**

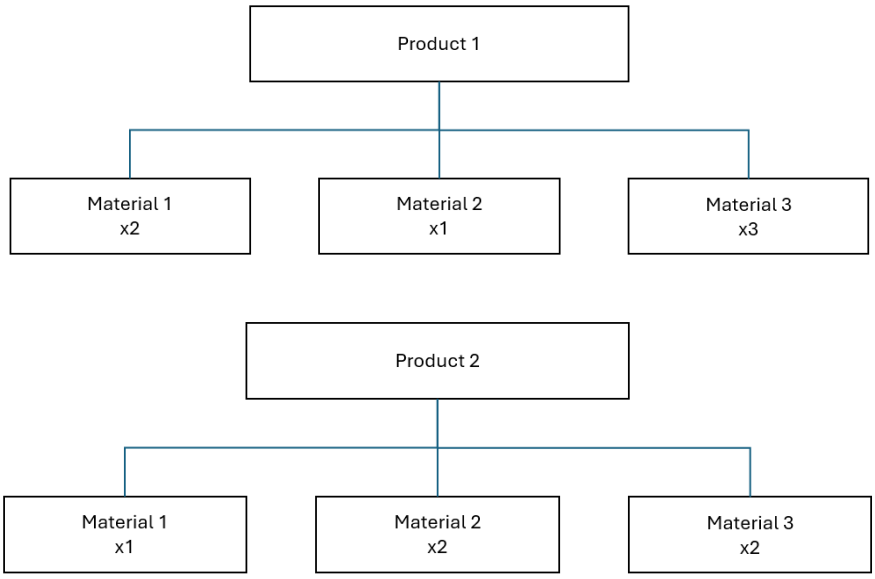
We will present a MATLAB code that considers a **single product** using *Q-learning*, and a Python code that considers the **multi-product** case using *Deep Q-learning*.

## 3 - Multi product DQL Product Assembly and Inventory Control

### 3.1 – Model and assumptions

In this more complex version of the model a multiproduct environment is considered.

The assembly line faces the production of two different products (p1 and p2) each of them requires a specific number of three different raw materials (m1,m2 and m3) according to its production recipe which are represented in the following bills of materials:




Inside the system backorders for the final products are allowed under a fixed maximum backorder level.

For the replenishment of the raw materials the lot purchase system has been replaced, in order to guarantee the stability of the algorithm, with a system similar to the widely used ROL(s,S) in which the reorder is performed when the level of stocks in the warehouse goes under the fixed reordering level s and a new order is issued by means of a variable quantity that is capable of restoring the maximum stock level S; in our case the agent choose autonomously based in its experience when to issue an order.

Also the model considers that with a small probability (p=0.005) even if the agent have choose to replenish a stock, this does not happen; this can simulate the effect of external issues among the supply chain that lead to a fail in the replenishment.

The costs incurred by the system are the classical ones from a standard inventory management context and are listed below:

1. Purchase cost : is the cost for purchasing the unit of raw material
2. Holding cost (HC): is the cost for holding one unit of product or material into the warehouse
3. Order cost (OC): is the cost for issuing an order, independently from the quantities
4. Backorder cost (BC): is the cost for keeping a single product in a backorder state

The demands for the final products are independent and different, they depends on the selling price chosen by the agent and on the previous level of sales; those prices are chosen from a discrete set.

Below a representation of the states and actions of the model:

States:

Each state is made by a vector of 9 components:

- Inventory level of m1
- Inventory level of m2
- Inventory level of m3
- Inventory level of p1
- Inventory level of p2
- Backorder level of p1
- backorder level of p2
- previous sales of p1
- previous sales of p2

actions

at each stage the agent takes the following decisions:

- replenish or not replenish material i (for i={1,2,3})
- select production of product 1 from a discrete set
- select production of product 2 from a discrete set
- select the price of product 1 from a discrete set
- select the price of product 2 from a discrete set

**3.2 – Deep Q learning**

Due to the large number of state and action of the model, a new different approach to deal with multiproduct was needed to implement it.

Deep Q learning combines the Q learning approach with neural network that are used to approximate the Q-function instead of maintaining a Q-table.

The network takes the state as input and returns the estimated Q values for all the possible networks, (?)this values are updated during the training of the network.

Two phases can be identified inside the deep Q learning process:

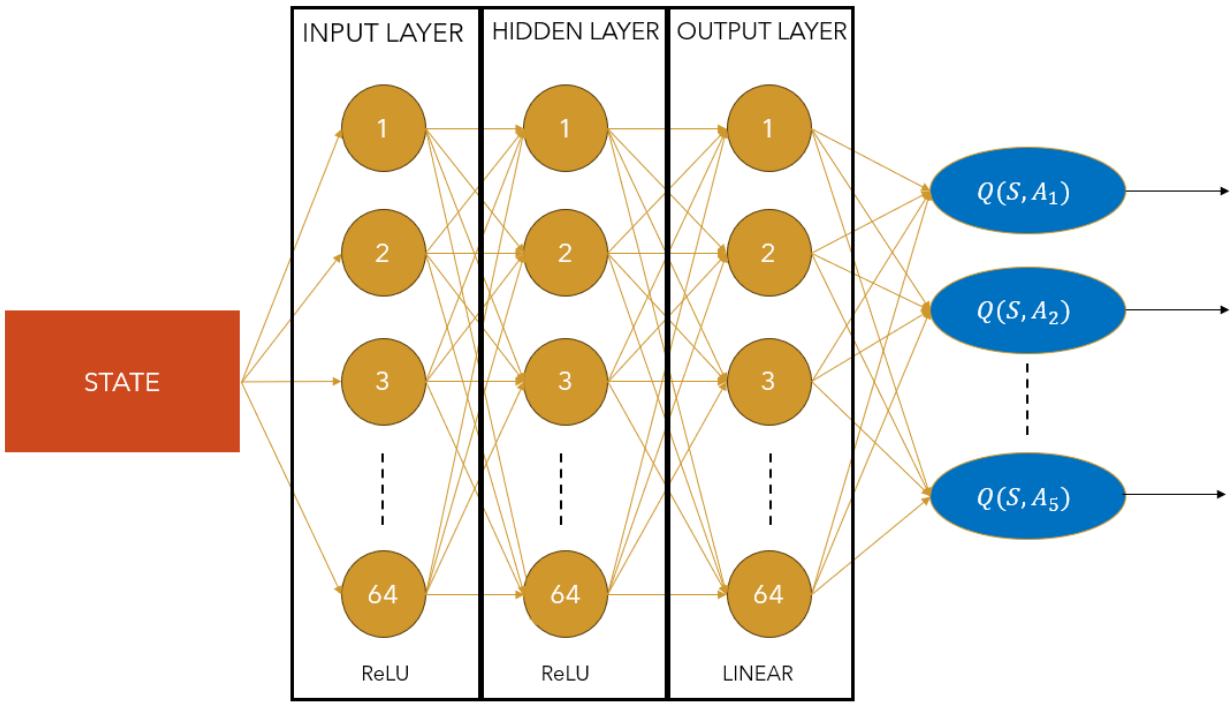
1) Data collection: the experience replay buffer is populated by the reward obtained by performing an action chosen with the epsilon greedy method form a given state
2) Training the DQN: select a random batch of data from the buffer, pass the batch into the Q network and pass it through the target network, then the computation of the Mean Square Error is performed over this two values and the result is backpropagated into the network and its parameters are updated

In this model a simple fully connected neural network composed by three layers each made by 64 neurons is implemented:

- Input layer: nonlinear, Rectified linear unit (ReLU)
- Hidden layers: a single nonlinear ReLU layer
- Output layer: linear layer

ReLU are used to add non linearity and give the network the ability to follow complex patterns.

Below a representation of the implemented network:




### 3.3 – Implementation

The code was developed in the python environment using the well-known open source library Pytorch; it spreadly uses class and functions to define the environment's dynamics and the Q network.

Below a brief analysis of the most important components of the code:

1. Environment dynamics and reward definition:

   Defined through the class InventoryManagementEnv in which the action space is defined, a reset function randomly initialize the state at the beginning of each episode and a step function defines for a given state action pair the next state and the reward associated with the action taken.

2. DQN definition:

   A class DQN containing a Depp Q Network as described above is initialized

3. Training loop

   For a fixed number of episodes (trials) each made by 360 days, the algorithm choose an action according to a epsilon greedy policy using the estimation of Q given by the network in the exploitation phase, then the next state and the reward are computed through the step function and all the information learned from the experience of the episode are added to the memory replay buffer for future training of the DQN.

   If the data inside the buffer are enough, a random sample of fixed size (batch) is extracted and is passed through the Q network and the values of the selected actions are gathered.

   By using the target network the maximum possible Q-values for the next states are computed and then the target Q-values are computed as the reward tensor plus gamma multiplied by those maximum Q-values.

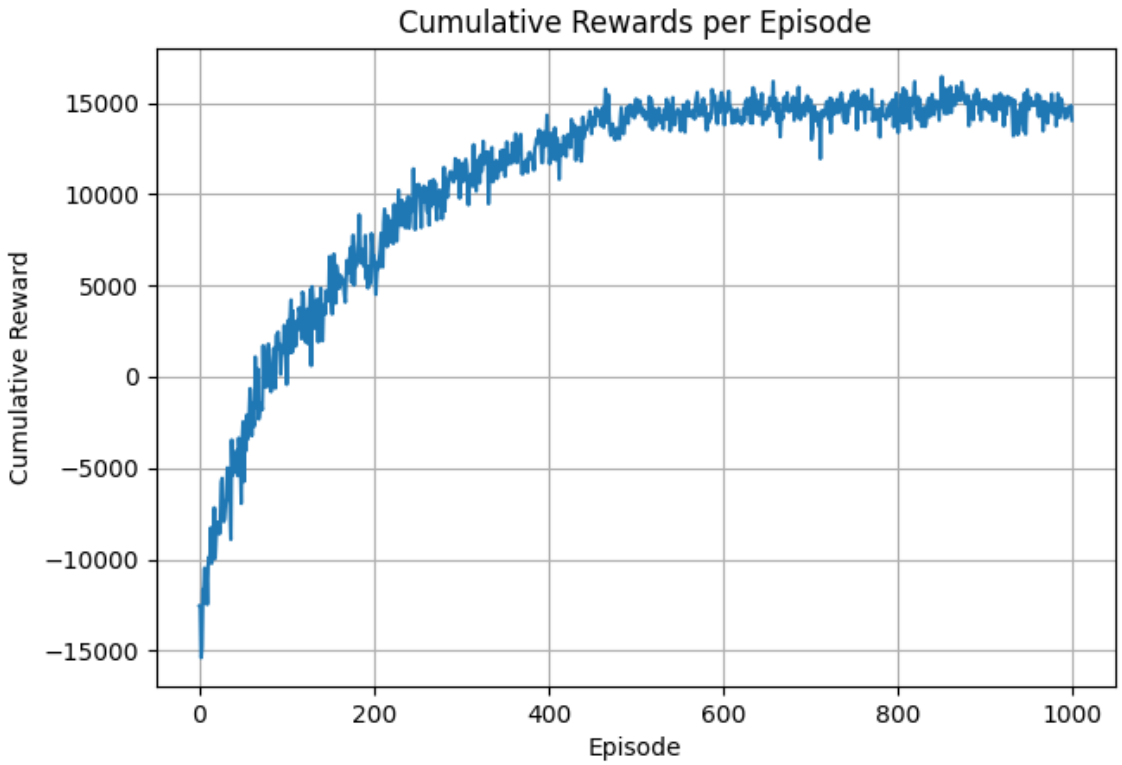
   A loss is calculated using the Mean Square Error between the Q network and the target network that is used to update the weights inside the Q network through the Stochastic gradient descent.

   Finally after each episode epsilon is updated and also each 10 episodes the target network is updated.

One of the main issues that needed to be addressed was the running time of the algorithm: While the model can handle a large number of states efficiently, the performance degrades significantly as the action space increases. This slowdown is primarily due to the computational burden associated with the two max operations: one during the action selection phase and another during the target Q-value computation. Both operations require evaluating the Q-values over the entire action space. As the action space grows, these max operations become increasingly costly, leading to a substantial increase in runtime, The proposed solution to reduce the runtime was to directly reduce the action space by reducing the set of prices and by implementing a ROL(s,S) reorder system.

### 3.4 – Results analysis

The results obtained by considering a time horizon of 360 days for each episode reported in the graph below, shows clearly a convergence after 500 episodes to an average value around 15000 from an initial value under -15000, which means that the agent was able to autonomously learn and increase the profits to a positive value through an optimal policy




The optimal policy demonstrates that, over time, the system minimizes product inventories (in respect to a maximum allowed value of 20) and meets demand primarily through real-time production. This behavior indicates that the system has autonomously implemented a Just-in-Time (JIT) production strategy, aimed at minimizing holding costs.

By producing goods only as demand arises and using raw materials immediately upon acquisition, the system effectively reduces the impact of holding cost (and minimizes backorder costs) since materials that are utilized for production within the same time period as they are replenished do not contribute to inventory holding costs, further enhancing cost efficiency.

### 3.5 – Pros and cons

The main advantages of this approach are the ability to easily manage multiple products and raw materials and the code's flexibility given by the wide use of classes and functions

while its drawbacks relays on the complexity of the algorithm  and the assumption that were necessary to avoid excessively long running times

## 4. Conclusions

In conclusion of this project experience, as a group we are satisfied with the work done and the results obtained as we managed to bring together our ideas and implement them, achieving acceptable outcomes.