Can reinforcement learning efficiently pilot a random supply chain?

Paul Bonin
paul.bonin@mines-paristech.fr
Supervising teachers: Philippe BLANC, Sébastien TRAVADEL

Keywords: suppply chain, reinforcement learning, randomness

Abstract

Reinforcement Learning (RL) can be applied to supply chain management in order to optimize both stock costs (Kegenbekov and Jackson [1]) and finished products shortages. However, randomness must be taken into account. This study aims at evaluating the impact of supply chain randomness on RL-based agents and testing the resilience of the latter in a highly variable environment.

1. Introduction

Kegenbekov and Jackson [1] implemented an actor-critic RL algorithm to train an agent that outcompetes base stock policy profit by around 5%. However, the way they simulated their supply chain included very little randomness. Everything was determinist, apart from the demand distribution (Poisson law). This determinism is not quite in phase with the realities of on-ground supply chain management.

This research project aims at finding out to what extent reinforcement learning techniques can or cannot outperform standard base stock policy when an increasing part of randomness is implemented in the supply chain model. I introduced several types of random events:

- Machine breakdown during one timestep. This means that during one iteration, a given machine does not produce anything.
- Random delivery time between supply chain steps.
- Random demand following a Poisson law of parameter 80.

2. Reinforcement Learning

This article is largely based on RL theory and notations developed by Sutton and Barto [2].

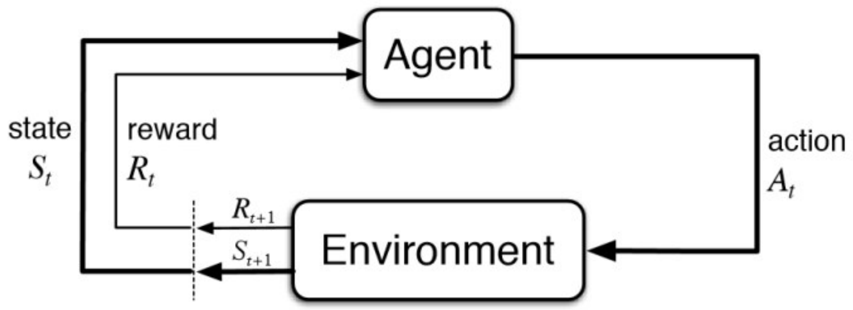

Figure 1: reinforcement learning pattern

In RL, an agent interacts with its environment through actions. At a given time $\boldsymbol{t}$ and given a state $\boldsymbol{S_t}$, the agent choses an action $\boldsymbol{A_t}$. This action transforms the environment into a new state $\boldsymbol{S_{t+1}}$, and moving from one state to another gives the agent a reward $\boldsymbol{R_{t+1}}$. The value of this reward is determined by a reward function that directly depicts the will of the programmer: if moving from state $\boldsymbol{S_t}$ to state $\boldsymbol{S_{t+1}}$ is something we wanted the agent to achieve, the reward is positive (or set to zero). However, in case of an unfavorable state evolution, the agent is penalized via a negative reward. The value of the reward is included in calculation when it comes to choosing the next action $\boldsymbol{A_{t+1}}$, and so on. This is summarized in **Figure 1**.

This study uses the q-learning technique to train the agent. For a given state $\boldsymbol{S_t}$**,** a q-value $\boldsymbol{q(S_t, A_t)}$ is computed for each possible action $\boldsymbol{A_t}$. This q-value represents the value of not only choosing action $\boldsymbol{A_t}$ in the state $\boldsymbol{S_t}$, but also of all the choices that will follow after action $\boldsymbol{A_t}$ is undergone in the state $\boldsymbol{S_t}$. In other words, q-values represent the sum of future reward following a given action in a given state, discounted by a $\boldsymbol{\gamma} \in [\boldsymbol{0}, \boldsymbol{1}]$ rate.

Q-values calculation algorithm is updated during the training phase of the agent, depending on the rewards $\boldsymbol{R_t}$ that are received when choosing different actions $\boldsymbol{A_t}$ in states $\boldsymbol{S_t}$. It is explained in detail in **3.5 The agent training**.

At each time period $\boldsymbol{t}$, the model simulates one iteration. An iteration lasts for one time unit. An iteration begins by considering the state $\boldsymbol{S_t}$ of the environment. Considering $\boldsymbol{S_t}$, the agent choses an action $\boldsymbol{A_t}$. More details about the action vector are given in the next section: **3.4** . The environment evolves to a new state $\boldsymbol{S_{t+1}}$ and gives a reward $\boldsymbol{R_{t+1}}$. One episode consists in repeating this pattern a given number of iterations. In this paper, the episode length is set to 25 iterations.

3. The model

3.1 The supply chain

I decided to implement a simple supply chain, as depicted in **Figure 2**. In this supply chain, raw materials (wood logs) are sawed to produce planks, that are then sold in a plank store. The sawing process produces wooden residue, that is processed into paper, which is sold in a paper store.

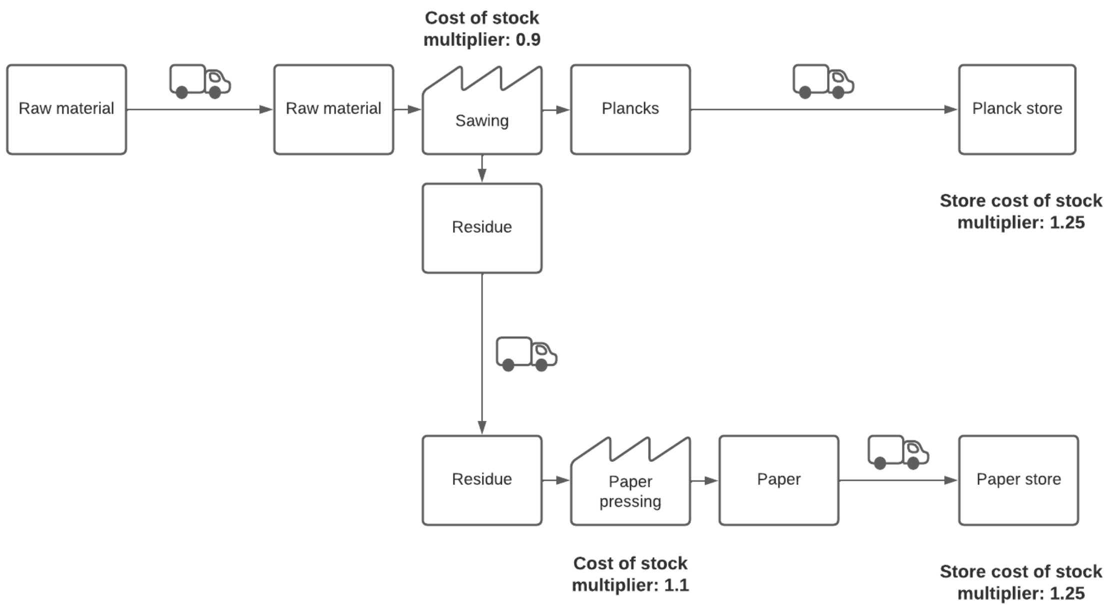
Cost of stock multiplier: 0.9
Raw material
Raw material
Sawing
Plancks
Planck store
Store cost of stock multiplier: 1.25
Residue
Residue
Paper pressing
Paper
Paper store
Cost of stock multiplier: 1.1
Store cost of stock multiplier: 1.25

Figure 2: modelized supply chain

Rectangles represent stocks, as opposed to sawtooth boxes that represent industrial processes (“sawing”, “paper pressing”). It has been assumed that this supply chain is made up of five different warehouses (raw material stock, sawing process factory, paper pressing factory, plank store, paper store). Each warehouse is either composed of a single stock (raw material stock, plank store, paper store); or of a combination of one process upstream stock, one process, and one or two process downstream stocks. Those warehouses are separated by trucks in **Figure 2**.

An upstream stock contains items that are waiting to be processed. A downstream stock contains processed items. “Sawing” process has two downstream stocks, each of which are filled by one unit every time one unit of “raw material” is processed. “Paper pressing” process only has one downstream stock. Units are processed one by one. The raw material stock is unlimited. Processes cannot process more than 100 units / iteration. All stocks have illimited capacities.

Stores can be assimilated to stocks. At the end of each iteration, items are sold and removed from store stock according to demand. If the demand is superior to the available store stock, the unmatched demand is considered “lost sales”. Demand follows a Poisson law of parameter 80.

Trucks represent transportation between stocks. Transportation costs are not considered in this paper. Transportation duration (in iterations) follows a Poisson law which parameter varies depending on simulations (0, 1 or 2). Transportation capacities are assumed to be unlimited.

At the beginning of each simulation, orders are passed by supply chain links to upstream links. Placing an order means ordering that a given number of items are taken out of a given stock in the supply chain to be moved to another stock, in order to process or sell these items. For instance, if the “paper pressing” process places an order of 120 items, 120 items will be taken out of the “sawing” process downstream “residues” stock and will be transported to the "paper pressing" process upstream "residues" stock. By doing so, items move from one warehouse to another down the supply chain.

3.2 Reinforcement learning elements

The state vector of the environment is defined according to the values (integer) of each of the seven stocks in the system (excluding the illimited raw material stock): $\boldsymbol{State\ vector} = [\boldsymbol{stock_1}, \dots, \boldsymbol{stock_7}]$. Those values are normalized following the rule: $\boldsymbol{Normalized\ state\ vector} = \boldsymbol{State\ vector} \,/\, \max_{\boldsymbol{i}} \boldsymbol{stock_i}$.

The action of the agent is a vector that dictates the quantity that each element of the supply chain (i.e. the two processes and the two stores) will order. This vector is of length four, because only four supply chain links place orders: the two processes and the two stores. For the sake of neural network simplicity, the orderable quantities are discretized with a step of 20, and range between 0 and 180. Keeping in mind that processes maximum capacities are set to 100, it means that the agent could decide to order more than what can be processed to create security stocks. The order of the vector is the following: [sawing process, paper process, plank store, paper store].

The neural network is described in **Figure 3**. The $\boldsymbol{Normalized\ state\ vector}$ is the input of the network. This input information is processed in three dense layers. The output of the third dense layer is then used four times to create four independent subnetworks. Each subnetwork gives out ten q-values. Those are the q-values associated with each order possibility (0, 20, …, 180), for each of the four supply chain links that pass orders (sawing process, paper process, plank store, paper store).

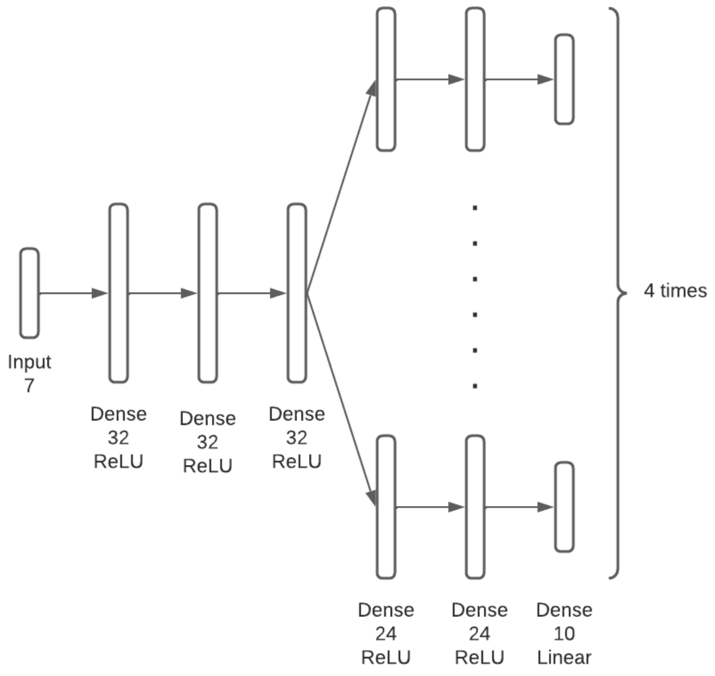

Figure 3: neural network

Profit is computed as: $\boldsymbol{Economic\ profit\ =\ profit\ on\ sales - cost\ of\ stocks}$. Stock penalties are computed at every iteration, which means that at the end of each time unit, every item presence in the supply chain is penalized. To reflect economies of scales, some stocks have higher unitary cost than others. It is the point of the “cost of stock multiplier” in **Figure 2**. The stock penalty of each stock at the end of each iteration is computed as follows: $\boldsymbol{Sock\ penalty\ =\ number\ of\ items\ in\ the\ stock \times cost\ of\ stock\ multiplier}$. The total cost of supply chain stocks is the sum of all stock penalties.

However, the reward function is separate from the economic profit. It indeed emphases stock penalties and lost sale penalties. Lost sale penalties intend to take into account long term brand damage caused by stock shortages. They are computed as follows: $\boldsymbol{number\ of\ missing\ items\ in\ stock\ \times unitary\ missed\ sale\ penalty}$. The unitary missed sale penalty is a parameter that depends on simulations. The reward function was designed to seek both supply chain efficiency (i.e. bear as little stock as possible) and finished products availability. Reward is calculated as: $\boldsymbol{Reward\ =\ -(total\ cost\ of\ stocks + lost\ sales\ penalties)}$, and is always a negative number. The agent seeks to maximize the reward, i.e., minimizing both stock and lost sales penalties.

3.3 Q-values and losses

Once an action vector is determined by the agent from the neural network q-values output, an episode is ran (one iteration, see **3.4 Running an iteration**), at the end of which a reward is computed.

The four q-values associated with the four chosen order amounts in the action vector are then updated: $\boldsymbol{q_{new}(S_t, A_t) = reward_t + \gamma \times \max_{A}(S_{t+1}, A)}$. $\boldsymbol{\gamma}$ is a model hyperparameter. It has been set to 0.6 in the following results since it’s the value that gave the best convergence when training the network.

For each of those four q-values, a loss is computed: $\boldsymbol{loss = (former\ q - value - new\ q - value)^2}$. The global model loss is the sum of those four sub-losses.

3.4 Running an iteration

Each iteration follows the same pattern. The environment is initially in a given state $\boldsymbol{S_t}$. Considering $\boldsymbol{S_t}$, the agent choses $\boldsymbol{A_t}$, which contains the orders indications for the two processes and two stores. Orders are immediately placed, which means that items are boarded onto trucks for transportation. Orders from previous iterations are then received, and stocks filled in. Processes instantaneously process items from their direct upstream stocks (in the limit of 100) to their direct downstream stock, from which they will then be shipped during another iteration. Store instantaneously sale their stock according to demand in the limit of available stock. The supply chain has then reached the new state $\boldsymbol{S_{t+1}}$.

Profit on sales, cost of stock and cost of missed sales are computed. The reward is computed, and neural network updated.

3.5 The agent training

For each new environment, the agent is trained from scratch during 2000 episodes of 25 iterations. 1 iteration = 1 time unit. The environment is reset at the beginning of each episode. Mini batches of 8 iterations are randomly picked up among the last 100 iterations simulated. A gradient descent is performed on the average loss of the mini batch. At each step, the idea is to bring $\boldsymbol{former\ q\ value}$ closer to $\boldsymbol{new\ q\ value}$ by shifting neural network weights.

4. Simulations

The results shown below are those of four simulations. The simulation parameters are:

- delivery Poisson law parameter (0, 1 or 2);
- process breakdown probability per iteration (0%, 20% or 30%);
- unitary missed sale penalty (10 or 20).

The performances of the reinforcement-trained agent are compared to:

- a random agent, which order instructions are randomly picked in $[0,20, \dots 160,180]^4$ following a uniform distribution;
- a basic agent which instructions consist of constantly ordering the same number of items at each step, namely the average of demand: $[80,80,80,80]$, no matter the state of the environment.

The parameters of interest are: financial gain, lost sales.

After being trained, RL-based agents are tested during 500 episodes of length 25. At the end of each episode, the average financial gain, and the average lost sales over the 25 iterations are computed (cf. “money_revenue” and “missed_sales” graphs in **5. Results**). It is then plotted and compared to the performances of the random and basic agents.

5. Results

5.1Simulation 1

<u>Simulation parameters:</u>

- Poisson delivery parameter: 0 (instantaneous)
- No random process breakdown
- Unitary missed sale penalty: 10

Without any randomness in the supply chain apart from demand (Poisson law, parameter 80), reinforcement-trained agent performs well. It performs as well as the “basic” agent.

By looking deeper into the results, it turns out that the agent manages to learn accurately the average of demand. In other words, the reinforcement-trained agent behaves exactly as the basic agent and orders $[80,80,80,80]$ no matter the state of the environment. Hence the results shown in **Figure 4**.

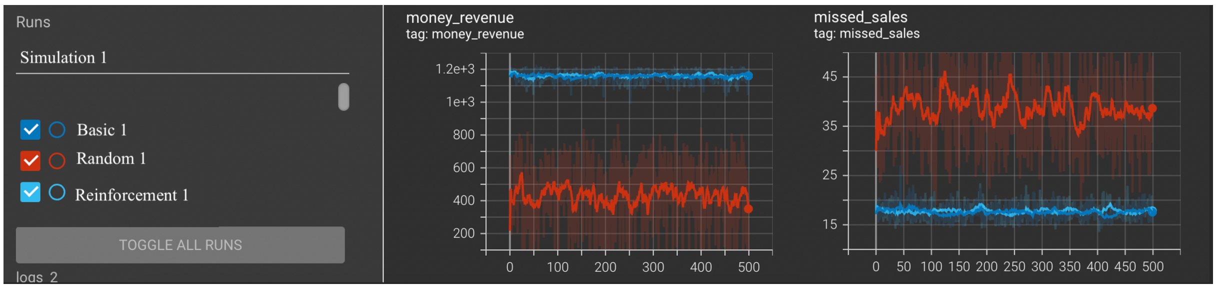

Figure 4: simulation 1 results

5.2 Simulation 2

<u>Simulation parameters:</u>

- Poisson delivery parameter: 1
- Process breakdown probability per iteration: 20% (uniform random)
- Unitary missed sale penalty: 10

When introducing randomness, reinforcement-trained agent outperforms the basic agent in terms of financial performances, thanks to decreased stock costs. However, this comes at an undesirable cost: increased missed sales, as shown in **Figure 5**. Even though this arbitrage does allow higher short-term profit, having a lot of item shortages is not sustainable in the long run from an industrial point of view, insofar as it would turn clients away from the company.

In terms of agent actions, the reinforcement-trained agent behaves once again deterministically. The action vector is $[60,120,180,120]$, no matter the state of the environment. If we dive into explanations:

- 60 in the first instruction means that at every iteration, 60 raw material items are ordered by the sawing process from the raw material stock (see **Figure 2: modelized supply chain** for a reminder). Therefore, the supply chain sales will be limited to 60 items per iteration per item class on average, when the average demand is set to 80.
- In case of a machine breakdown, stocks will accumulate before the faulty machine in the supply chain. Setting high order amounts for the other instructions (120, 180) allows those items to rapidly flow through the supply chain during the iteration that follows the machine breakdown.

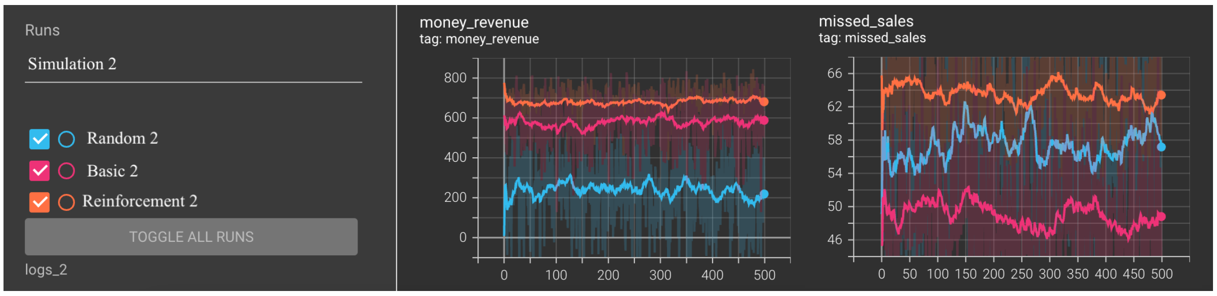

Figure 5: simulation 2 results

5.3 Simulation 3

Simulation parameters:

- Poisson delivery parameter: 1
- Process breakdown probability per iteration: 20% (uniform random)
- Unitary missed sale penalty: 20

The unitary missed sale penalty of 10 used in **5.2 Simulation 2** led the reinforcement-trained agent to purposely underproduce to optimize stock costs, which led to an undesirable increase in missed sales. In order to counter this tendency, this simulation uses the same parameters apart from the unitary missed sale penalty that is here doubled to 20.

This time, the reinforcement-trained agent misses less sales and financially outperforms other agents, as shown in **Figure 6**.

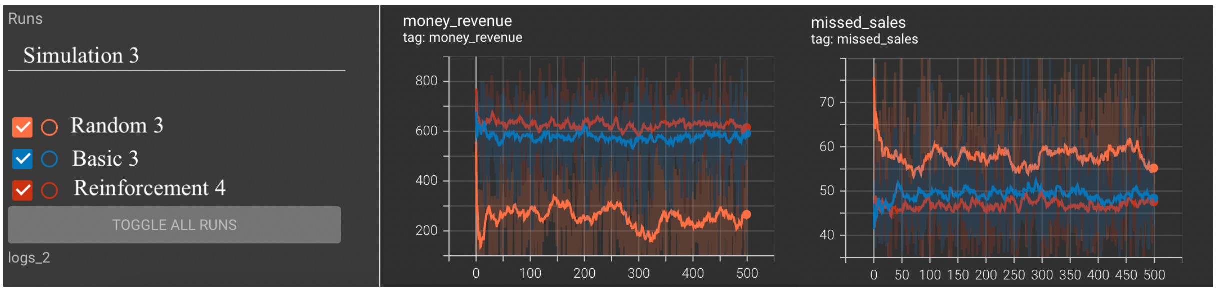

Figure 6: simulation 3

5.4 Simulation 4

Simulation parameters:

- Poisson delivery parameter: 2
- Process breakdown probability per iteration: 30% (uniform random)
- Unitary missed sale penalty: 20

Considering **5.3 Simulation 3** satisfying results, I decided to further challenge the reinforcement agent by increasing randomness in Poisson delivery and process breakdowns. However, the simulation outcome turned out to be like **5.2 Simulation 2**. Even though the reinforcement-trained agent achieves the highest profit, it misses far more sales than both other random agents.

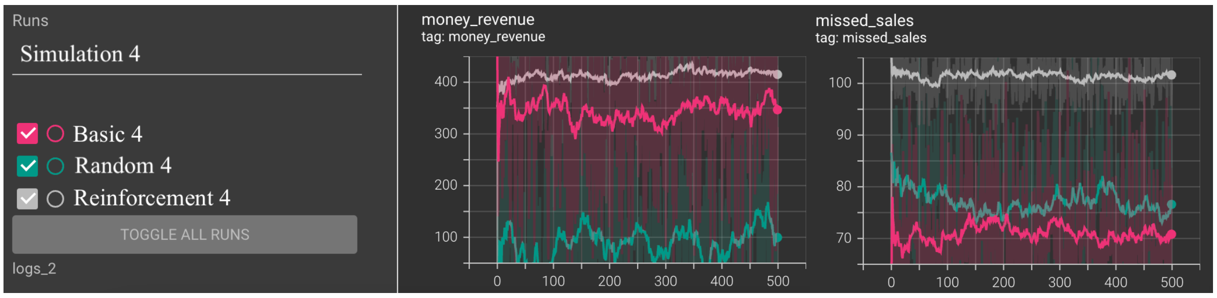

Figure 7: simulation 4

6. Conclusion

All in all, our RL-based supply chain management policy has proven capable of slightly outperforming a basic policy that would aim at producing at the average of demand in a random context. However, agent reward parameters must be carefully tuned, to avoid the undesirable tendency previously observed in **5.2 Simulation 2** to scarify production and store products availability to reduce stock costs.

The supply chain that was first modelized for this paper was twice as complex as this one (see **3.1 The supply chain** for a reminder) and would produce four kinds of products. The model, when applied on this more complex supply chain, would not converge, hence a simplification to obtain some results. Therefore, the implementation of such an algorithm on real-scale industrial supply chains remains hazardous, due to the high level of complexity of the latter, not to mention the difficulty of on-ground implementation.

Thanks

Special thanks to Mathis Bourdin for his help throughout my work.

- https://www.linkedin.com/in/mathis-bourdin-33201a22a/
- https://github.com/mathisbrdn/

Bibliography

[1] Kegenbekov Z and Jackson I 2021 Adaptive Supply Chain: Demand–Supply Synchronization Using Deep Reinforcement Learning *Algorithms* **14** 240

[2] Sutton R S and Barto A G 2018 *Reinforcement Learning, second edition: An Introduction* (MIT Press)