Table of Contents

Introduction

In this report, we will be focusing on a dynamic inventory routing a problem that consists of 19 Jumbo stores and 1 warehouse where the stores need to be replenished from. Each day, a decision must be made about which stores will be replenished and up to which inventory level they will be replenished considering that each store has a maximum capacity of 1000. A figure of the visualization of the problem is provided as below.

Figure 1. Visualization of the Problem

There are three types of costs associated with the problem: Costs associated with replenishing the stores, holding cost of excess inventory at the end of the day, and loss sales in case of a stock-out situation.

The cost associated with transporting goods to the stores depends on a number of variables such as a fixed cost for using a truck, total distance covered during the transportation and the number of trucks used to replenish the stores. In our case, there are 19 trucks available with a vehicle capacity of 100 each day. The coordinates of the warehouse and the stores are provided in the problem allowing us to compute the actual distance among the stores and the warehouse.

In addition to the costs associated with the transportation of goods, holding cost for left-over inventory will be 1 per inventory and the cost of loss sales in stock-out situation will be 19 per day for each item.

In this problem, our overall goal is to minimize the total cost of the system by training a DRL method. We will start by formally formulating the MDP of the problem, then, we will continue with preparing and improving the gym environment for training. Once we get the environment ready for training, we will start training our models.

MDP Formulation

States

We will formulize the states in our problem as an array consisting of the inventory levels of each store at the end of the day. To write formally:

$$s \in S_t = \left[I_t^{(1)}, I_t^{(2)}, I_t^{(3)}, \ldots, I_t^{(17)}, I_t^{(18)}, I_t^{(19)}\right]$$

where $I_t^{(i)}$ represents the inventory level of store i at a particular time t.

Actions

Dealing with the action space is one of the most important aspects of our problem. Throughout this report, we will come up with different formulations for the actions depending on our approach and policy. Since different models can have different ways of formulating the action space, we will leave the detailed formulation of the action space to the sections where the models will be introduced. However, if no assumptions or approximations are made about the action space, the action can be formulated as the replenishment amounts for each store. More formally:

$$a \in A_t = \left[R_t^{(1)}, R_t^{(2)}, R_t^{(3)}, \ldots, R_t^{(17)}, R_t^{(18)}, R_t^{(19)}\right]$$

where $R_t^{(i)}$ represents the replenishment amount for store i at a particular time t. It is important to mention that there are 19 truck available each with a maximum capacity of 100. Therefore, the total replenishment in a day cannot be higher than 1900.

Rewards

As explained in the introduction section, there three types of costs in the system:

1. Transportation Cost

The transportation cost consists of a fixed transportation cost of using a truck to replenish to store per truck and a unit transportation cost depending on the distance covered. The fixed cost of using is a truck is 20 per truck and the unit cost is 2.5 per km. Using this information, the transportation cost can be formulated as:

$$Transportation\ Cost_t = \#\ of\ Trucks\ used_t \times -20\ + Distance\ Covered_t \times -2.5$$

A vehicle routing solver is provided in the gym environment to determine the cheapest path to choose given the action.

2. Holding Cost

Holding cost is 1 for each excess inventory that are not sold at the end of the day. It can be formulated as:

$$Holding\ Cost_t = -1 \times \sum_{i=1}^{19} I_t^{(i)}$$

3. Loss Sales Cost

Loss sales cost is 19 for each sale that are not fulfilled due to stock-outs. It can be formulated as:

$$Loss\ Sales\ Cost_t = -19 \times \sum_{i=1}^{19} Loss\ Sales_t^{(i)}$$

Combining the costs mentioned above, total reward at a particular time t can be formulated as:

$$Total\ Reward_t = Transportation\ Cost_t + Holding\ Cost_t + Loss\ Sales\ Cost_t$$

State Transitions

After taking a particular action in a particular state, the stores are replenished with the amounts specified in the action. After the replenishment a stochastic demand coming from a normal distribution is realized for each store. The means of the demand distribution are 4 for stores 2, 7, 11, 13, 16, 17, 19; 10 for stores 1, 3, 4, 12, 14, 15, 18 and 25 for stores 5, 6, 8, 9, 10. The standard deviation for the normal distribution for each store is then calculated as:

$$\sigma^i = \lceil 0.5 \times \mu^i \times random(0,1) \rceil\ for\ each\ integer\ i \in [1,19]$$

$$D_t = [\mathcal{N}_t(\mu^1, \sigma^1), \mathcal{N}_t(\mu^2, \sigma^2), \ldots, \mathcal{N}_t(\mu^{19}, \sigma^{19})]$$

where $random(0,20)$ returns a random number between 0 and 1. Demands are then sampled from the resulting distribution in each time step t. The resulting state transitions after demands are realized can be formulated as:

$$S_{t+1} = max(0, \quad min(1000, S_t + R_t) - D_t)$$

GYM Environment

Detailed explanations about the gym environment's functionality are given in our code submission; however, we would like to mention some important functionality and changes we made to the environment to make it work better.

The overall environment consists of three core elements: initialization, reset function, and step function.

Initialization

In this part, we initialize the observation and action space that are needed to define the environment. While the observation space is the same for all the models we are going to mention in the next sections, the actions space is different depending on how we deal with the action space.

Additional to observation and action space, the fixed parameters such as number of vehicles available, holding and loss sales cost, transportation costs, demand means and standard deviations, and the x, y coordinates of the stores and the warehouse are initialized in this part of the environment. The distance matrix that stores the cost of transportation is also calculated and stored in this part. We also added a copy of the distance matrix to this part for reasons we will mention later.

Step Function

The step function is the core element of the environment. It takes an action, calculates the transportation cost using the *calcDirectReward* function provided, samples a demand distribution, and realized the new state along with the holding and loss sales costs. It then returns the new state, total reward, and the boolean value of whether or not the episode ended. Since there is no particular end to the episodes, we will use episodes consisting of 20 days and finish the episodes after 20th day. Step function uses two very important helper functions while performing those calculations:

> **1. calcDirectReward:** This function takes an action and calculates the transportation cost using a module called *hygese*. We have made some crucial changes to the original code provided to us in this part since it was not working properly initially. The module was programmed to visit each store even though there were no replenishment to be made. To avoid that, we used the copy of the distance matrix to filter so that only have warehouse and the stores that are going to be replenished at each time step. We also filtered other parameters *hygese* module uses to match the dimensions. By doing so, we were able to make the model only visit the stores that are going to be replenished.

Another important problem about the *hygese* module was the fact that the edge cases were not covered. The module was throwing "NULL Pointer Error" in the cases when no stores or only 1 store were replenished. We have also written basic if else statements to avoid these cases.

Another issue was about the time complexity of the *hygese* solver and its effect on the training times. To speed up to process, we used a dictionary as a lookup table. The dictionary used was consisting of binary values for each store, indicating whether or not the store is to be replenished. It is constructed by basically caching the results as we trained the model. By doing so, we were able to approximate the transportation cost and speed up our training model significantly. Since the dictionary was consisting of approximate values, we only used it during the training, and used *hygese* solver while testing the results. The dictionary can be found in the "lookup.pkl" file in our submission.

2. _take_action: This helper function was used to take the action, update the inventories, filter the parameters of the hygese solver. We implemented our policies inside this function. Therefore, although most of the environment will stay the same in different models, this code inside this function is likely to change in each other model we try.

Reset Function

Reset function was used the reset the parameters of the environment such as current step and the inventories after the end of each episode. In our setting, we decided to change the starting inventory from 0 to randomly picked inventory levels between 0 and 10 to create a more realistic setting.

Analytics Function

Analytics function returns the basic statistics such as transportation cost, holding cost, and the loss sales. It is used to evaluate the model after training.

Benchmark Solution

As a benchmark solution, we have trained a REINFORCE algorithm with a simplified action space and a (s, S) policy. We have simplified the action by restricting the action to only indicate how many stores should be replenished each day. We have trained the model to output the number of stores to replenish given the current inventory level. After number of stores to replenish are decided, we sorted the stores based on how far they are from their order-up-to levels and took the ones that are furthest. After deciding on the stores to replenish, we looked up their inventory levels to decide on the replenishment amount. For the stores whose inventory levels are smaller than the order-up-to levels, we calculated the order amount as the difference between the current inventory level and order-up-to level times two. The formal definition of the action taking process is as below.

Actions

After trained REINFORCE algorithm gives an output indicating the number of stores to replenish, this output is used to get the stores that are furthest away from their order-up-levels. If the model's output at time t is denoted by $\mathcal{N}_t$, the decision-making pipeline can be formulated as:

$$inventory\ deficit_t^{(i)} = inventory_t^{(i)} - order - up - to\ level^{(i)}$$

$$stores\ to\ replenish_t^{(i)} = inventory\ deficit_t^{(i)}.argsort()[:\mathcal{N}_t]$$

After the stores to replenish are decided, the stores that have less inventory than their order-up-to levels are replenished to a level of order-up-to level times two. With the idea of not having to replenish all stores every day and thus minimizing the transportation costs, we motivate this decision of ordering to a level of order-up-to level times two.

$$a \in A_t = \left[R_t^{(1)}, R_t^{(2)}, R_t^{(3)}, \ldots, R_t^{(17)}, R_t^{(18)}, R_t^{(19)}\right]$$

$$R_t^{(i)} = 2 \times order - up - to\ level^{(i)} - inventory_t^{(i)}\ \forall\ i \in [1, 19]\ and\ i\ is\ integer$$

Before analyzing the student, the general outline of the REINFORCE algorithm used to train the benchmark model will be explained.

REINFORCE Algorithm

Reinforce of algorithm one of the most popular policy-gradient methods used in reinforcement learning. Policy gradient algorithms uses the gradient of the expected rewards with respect to policy gradients and try to learn a parametrized policy instead of a state-value function like we did in Assignment 1. Additionally, REINFORCE algorithm outputs a probability distribution for the possible instead of Q-values. To output probabilities, it makes use of a softmax function in its output layer. The general steps involved in the algorithm are summarized below.

1. Initializing the parameters.
2. Generating a number of episodes by following a policy.
3. Computing the discounted sum of rewards of the episodes for each episode.
4. For each episode, computing the policy gradients by taking the product of the discounted sum of rewards and the gradient of the log probability of actions taken by following the policy.
5. Computing the average policy gradients for the number of episodes.
6. Updating the gradients using the gradients computed at 5 and multiplying by a learning rate that is determined previously.

Below is the pseudocode of the REINFORCE algorithm.

Algorithm 1 REINFORCE

Require: A differentiable policy parameterization $\pi(a \mid s, \theta)$
Initialize the parameters θ
Select step-size parameters $0 < \alpha \leq 1$
Choose discount rate $0 < \gamma \leq 1$
Choose max number of episodes N
Choose number of episodes to batch together for an update $K \geq 1$
while Episode $n < N$ **do**:
  for K batches **do**:
  Generate an episode $s_0, a_0, R_0, ..., s_t, a_t, r_t$ following policy $\pi(a \mid s, \theta)$
    for each step in the episode (t), discount rewards **do**:
    $G_t \leftarrow \sum_{t=1}^{T} \gamma^t R_t$
    end for
  Calculate policy loss for all episodes in the batch $\mathcal{L}(\theta) = -\frac{1}{m} \sum_t^T ln(G_t \pi(a_t \mid s_t, \theta))$
  Update the policy: $\theta \leftarrow \theta + \alpha \nabla \mathcal{L}(\theta)$
  Increment episode counter $n \leftarrow n + 1$
  end for
end while

Figure 2. pseudocode of the REINFORCE algorithm. (Source: towardsdatascience.com)

Results

We have trained our REINFORCE algorithm for 500 episodes, with a learning rate of 0.0001 and a batch size of 5 that we optimized. Below is the figure of the structure of the neural network we used for training.

```
# neural network
class PolicyNet(nn.Module):
    def __init__(self, env):
        super().__init__()
        # dimension of observation space, 20
        self.n_inputs = env.observation_space.shape[0]
        # number of outputs is 20, how many stores to replenish [0, 19]
        self.n_outputs = env.action_space.n
        # neural network structure
        self.network = nn.Sequential(
            nn.Linear(self.n_inputs, 256),
            nn.ReLU(),
            nn.Linear(256, 64),
            nn.ReLU(),
            nn.Linear(64, self.n_outputs),
            nn.Softmax(dim=-1))
    # given a state, returns the action probabilities
    def predict(self, state):
        action_probs = self.network(torch.FloatTensor(state).to(device))
        return action_probs
```

Figure 3. Structure of the Policy Network

After training the model for 500 episodes, below is the figure of the learning curve with number of episodes on the x axis and the average rewards in the y axis.

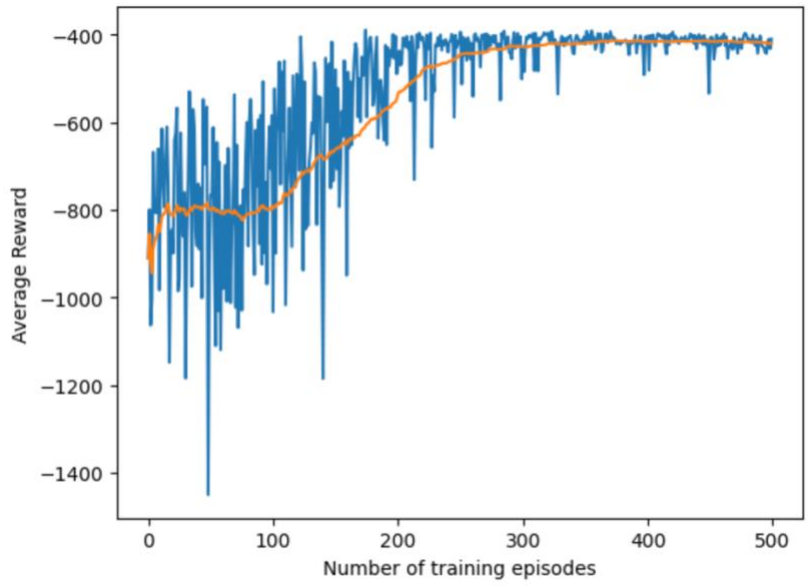


Figure 4. Training Curve

As can be seen from the above figure, our REINFORCE algorithm managed to converge to an average reward of around -420 after 200 episodes of training. A more detailed look into the actions taken by the model reveals that the model started to replenish 9 of the stores almost each day after some point. This result is understandable and expected since the motivation behind replenishing stores to a level of order-up-to level times two was to keep transportation

costs low. Below figure provides a deeper look into how the costs are distributed during the training.

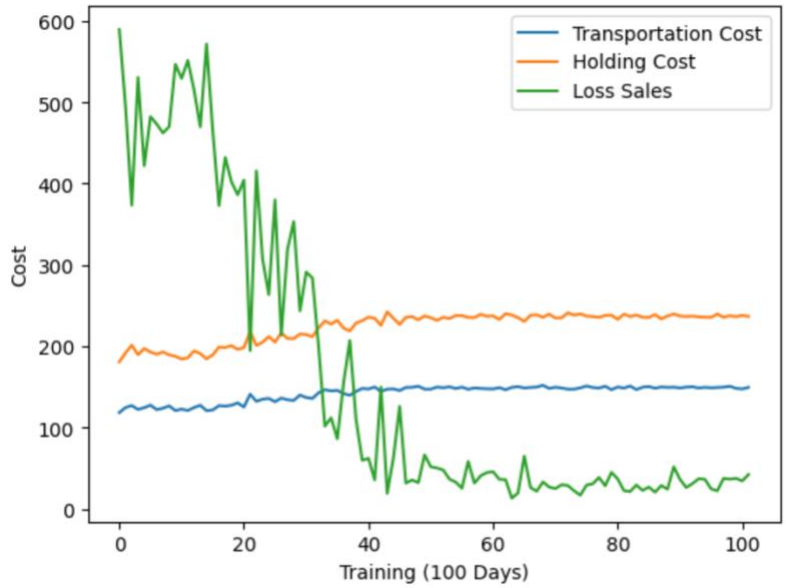


Figure 5. Distribution of Costs

Above figure shows that the model was able minimize loss sales to almost 0; however, with the expense of suffering a very high holding cost. Therefore, as an improvement to the benchmark solution, we decided to focus on finding a balance between the holding cost and loss sales.

Improvement 1 to the Benchmark Solution

The main issue we observed in the benchmark solution was a very high holding cost. While our model was able to learn how to minimize loss sales, it was not able to minimize the holding costs, partly because of the restrictions we set on the action space. As an improvement, therefore, we will give our model a partial flexibility by enlarging the action space by including an additional decision about up to what ratio of the order-up-to level stores should be replenished.

Since we are trying to minimize the holding cost, the model will have options to choose from 1.00, 1.25, 1.50, 1.75, 2.00. The ratio was set to be 2.00 by us in the benchmark solution and the model was converged to a solution where it ended up ordering for 9 stores in each day. To make action space smaller to reduce the training times; therefore, we will set a restriction about replenishing at least 5 stores each day. The resulting policy and action space is as formulized below.

Actions

After trained REINFORCE algorithm gives an output indicating the number of stores to replenish and up to what ratio of order-up-to level to replenish, this output is used to get the stores that are furthest away from their order-up-levels. If the model's output at time t is denoted by $Integer_t \in [0, 75]$, the decision-making pipeline can be formulated as:

$$Number\ of\ stores\ to\ replenish\ \mathcal{N}_t = Integer_t\ \%15\ +\ 5$$

$$\mathcal{N}_t, integer\ and \in [5, 19]$$

$$ratio_t = \lfloor Integer_t\ /\ 15 \rfloor / 4\ +\ 1$$

$$ratio_t \in [1.00, 1.25, 1.50, 1.75, 2.00]$$

$$inventory\ deficit_t^{(i)} = inventory_t^{(i)} - order - up - to\ level^{(i)}$$

$$stores\ to\ replenish_t^{(i)} = inventory\ deficit_t^{(i)}.argsort()[:\mathcal{N}_t]$$

After the stores to replenish are decided, the stores that have less inventory than their order-up-to levels are replenished to a level of order-up-to level times two. We motivate this decision of ordering to a level of $order - up - to\ level \times ratio$ by the idea of not having to replenish all the stores each day and therefore minimizing the transportation costs.

$$a \in A_t = \left[R_t^{(1)}, R_t^{(2)}, R_t^{(3)}, \ldots, R_t^{(17)}, R_t^{(18)}, R_t^{(19)}\right]$$

$$R_t^{(i)} = ratio_t \times order - up - to\ level^{(i)} - inventory_t^{(i)}\ \forall\ i \in [1, 19]\ and\ i\ is\ integer$$

Results

We have trained our REINFORCE algorithm for 500 episodes, with a learning rate of 0.001 and a batch size of 10 that we optimized. The structure of the neural network we used was the same as in Figure 3. After training the model for 500 episodes, below is the figure of the learning curve with number of episodes on the x axis and the average rewards in the y axis.

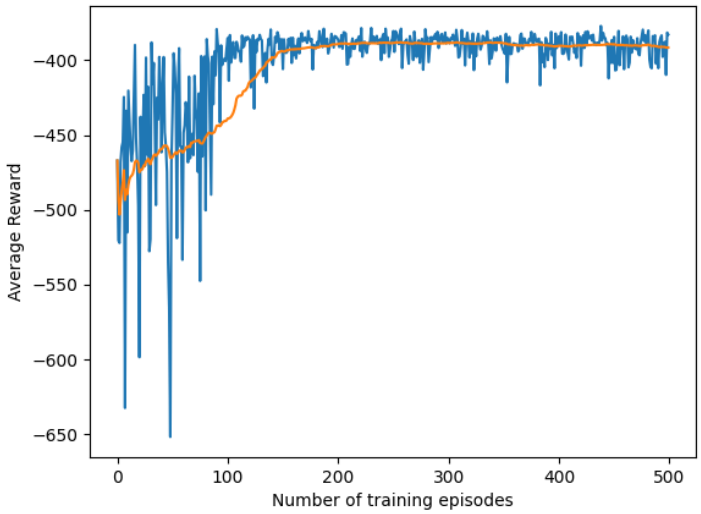


Figure 6. Training Curve for Improved Solution 1

As can be seen from the above figure, our REINFORCE algorithm was able to converge to an average reward of around -390 after 200 episodes of training. A more detailed look into the actions taken by the model reveals that the model started to replenish 18 of the stores to an order-up-to ratio 1.50. Although the ratio is understandable, the decision to replenish almost every store by replenishing 18 out of 19 stores to avoid possible sales loss situations can be a local optimum and subject to improvements with better policies and more detailed models. Below figure provides a deeper look into how the costs are distributed during the training.

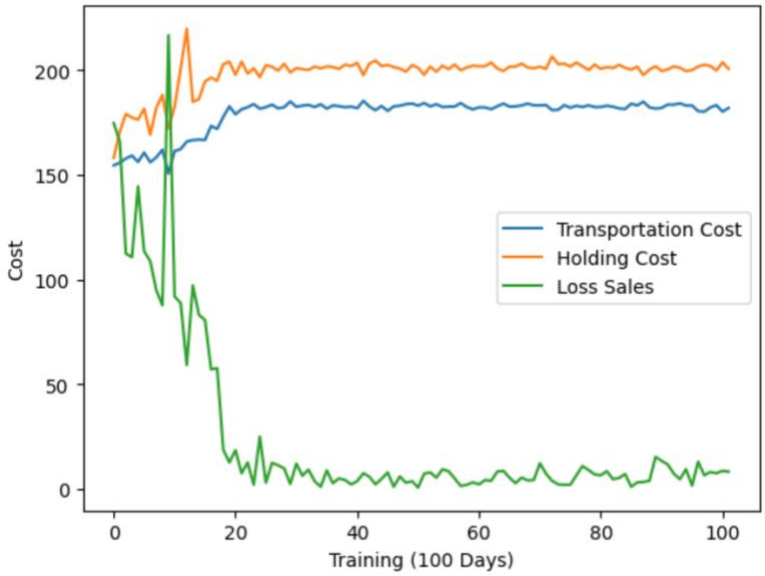


Figure 7. Distribution of Costs for Improved Solution 1

Below is a tabular comparison of benchmark solution and the first improved solution after 200-days of simulation.

	Transportation Cost	Holding Cost	Lost sales	Total
Benchmark	-149.22	-236.94	-37.77	-423.93
Improved Solution 1	-181.46	-201.67	-7.71	-390.84

Table 1. Comparison of Benchmark and Improved Solution

Our improved solution provided an improvement of more than 30 per day in general, with better holding cost and lost sales values. Due to the high number of stores to be replenished every day, however, it was worse in terms of the transportation cost. We will try to address the issue of transportation cost and high holding cost in our follow-up model.

Discussion Section on Failures

Before going to our final solution, we would like to discuss what worked for us and what did not. After realizing that our REINFORCE algorithm was prone to getting stuck in local optimums and producing unstable results from time to time, we decided to switch using a more state of the art approach called PPO (Proximal Policy Optimization) that is provided within the gym module and is another type of policy gradient method. In their paper on proximal policy optimization, (Schulman, et al. 2017) state that the new method they are proposing enables multiple epochs of minibatch updates whereas standard policy gradient methods such as REINFORCE perform one gradient update per data sample. PPO also uses clipping to take the biggest update step possible without getting too far from the old policy. This was one of the major reasons why we chose to switch to PPO from REINFORCE, to avoid our model from getting stuck at local optimums as much as possible. Another very important reason for the decision to switch was the compatibility of PPO gym spaces. PPO can handle Discrete, Box, MultiBinary, and MultiDicrete action spaces without changing a single line of code. Since we will be trying different action spaces in different models, we thought that the adaptability of PPO will help us a lot to save time during the training.

Failure 1

After switching from REINFORCE to PPO, the first thing we wanted to see was how much PPO can handle. To test it, we used a multi binary action space for the decision of replenishment of stores. We fixed the policy by using a (s, S) policy with

$$s\ (max.inventory\ level\ before\ ordering)\ =\ 0.8\ \times order-up-to\ level$$

$$S\ (max.inventory\ level\ after\ ordering)\ =\ 2\ \times order-up-to\ level$$

We hoped that that PPO would learn a replenishment pattern such that it effectively avoids replenishing stores when it's unnecessary and therefore minimize the transportation cost and holding cost while avoiding lost sales. The motivation behind this policy was to enable our model to learn to cover a region, for example, one day west and east the other day. After more than an hour of training with little to no improvement in episode reward means, however, we gave up and concluded that such action space ($2^{19} = 524288$) was too big for us and PPO to handle and get a feasible solution in reasonable amount of time.

Failure 2

After the first failure we encountered, we decided to shrink the action space by grouping the stores based on their locations. The first grouping we tried is given below.

Groups	Stores
Group 1	1-2-3-4-5
Group 2	6-7-8-9-10
Group 3	11
Group 4	12-13
Group 5	14
Group 6	15
Group 7	16-17
Group 8	18
Group 9	19

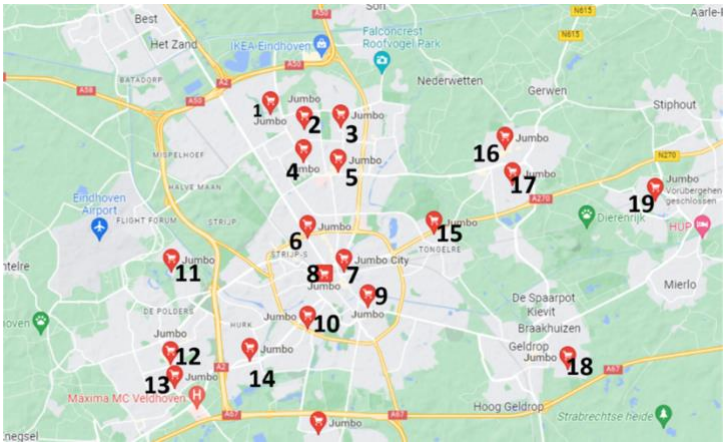


Table 2. Groups and Locations of the Stores

After grouping the stores, we decided to create a multi discrete action where the model have to choose between ordering nothing or ordering up to either 1.1 or 2.2 times the order up to level. Again, we motivate the choice of 1.1 and 2.2 by giving to model the freedom of learning a policy where it does not have to replenish stores each day that are far away from the warehouse and therefore costly to replenish. By doing so, we expected our model to learn a policy where both types of costs are minimized. After the model makes a decision to replenish a group, we wrote a code that will replenish the store up to an inventory level $(1.1\ or\ 2.2) \times order-up-to\ level\ of\ the\ store$ if the store's inventory level is less than $0.9 \times order-up-to\ level\ of\ the\ store$.

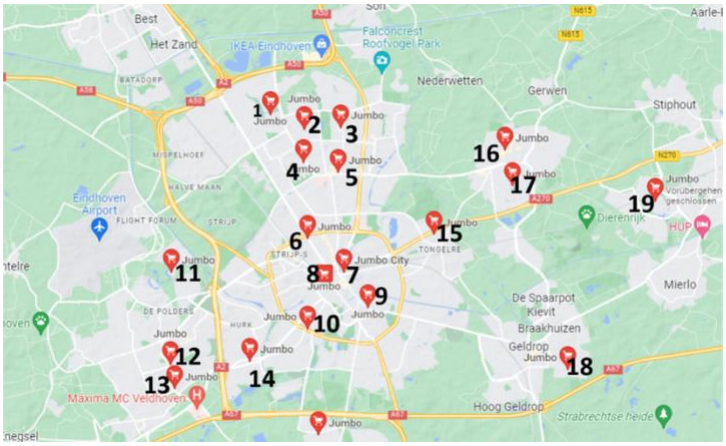
Again, after an hour of training, however, our model was stuck around an average reward of -500 per episode. We conclude that the action space of $3^9 = 19683$ is still very big and needs to be reduced.

Failure 3

After realizing that 9 groups are still too big of an action space for our models to give logical results in reasonable training times, we decided to shrink to number of groups to 7. Below is the new groups.

Groups	Stores
Group 1	1-2-3-4-5
Group 2	6-7-8-9-10
Group 3	11
Group 4	12-13-14
Group 5	15-16-17
Group 6	18
Group 7	19

Table 3. New Groups and Locations of the Stores



With the same configuration as in Failure 2 but this time with 7 groups and $3^7 = 2187$ possible actions, we trained a new model. The results were good with an average reward per episode of around -390 but the model was still struggling to improve more even though we also tried to optimize the hyperparameters such as number of epochs, batch size, learning rate, and discount factor. Although the results of the model were not as good as expected, the decisions taken by the model provided valuable insights on how we should carry on to further improve our model. The stores on group 1 and group 2 were rather close to each other and to the warehouse and therefore did not contribute to the transportation cost significantly. Therefore, we made a decision to replenish those stores every day up to an inventory level of $1.1 \times order - up - to\ level\ of\ the\ store$ if their inventories are below $0.9 \times order - up - to\ level\ of\ the\ store$. By doing so, we would allow our model to focus on how to minimize the rather high transportation cost of other stores that are further away from the warehouse and from each other.

Improvement 2 to the Benchmark Solution

As explained above, we trained our model to order every day for group 1 and 2 to an order-up-to level determined by us and let the model decide about up to what ratio of order-up-to level to replenish other stores.

Actions

After trained PPO algorithm gives an output indicating the number of stores to replenish and up to what ratio of order-up-to level to replenish, our model then replenished those stores if their current inventory level is less than $0.9 \times order-up-to\ level\ of\ the\ store$. The stores in the group 1 and group 2 are replenished every day to the inventory level of $1.1 \times order-up-to\ level\ of\ the\ store$ since we learned that their transportation cost is rather trivial because they are both close to the warehouse and close the each other. The crucial decision model makes is about the other groups, which are farther from the warehouses and farther from each other. The model output is formulated below.

$$Model\ Output\ =>\ list\ of\ (Group\ number, Replenishment\ Ratio)$$

$$Group\ number_t \in [1, 2, 3, 4, 5, 6, 7]$$

$$Replenishment\ Ratio_t \in [0.0, 1.1, 2.2]$$

$$inventory_t^{(i)} = inventory_t^{(i)} + Replenishment\ Ratio_t \times order-up-to\ level^{(i)}$$

$$(if\ inventory_t^{(i)} <\ 0.9 \times order-up-to\ level^{(i)})$$

Results

We trained our model for 5000 episodes, with optimized hyperparameters of a learning rate of 0.0005 and a batch size of 20. Below is a figure depicting the distribution of costs during the training. Please note that the transportation cost here is an approximation and it may not reflect the actual values. The real values obtained from the solver during the testing phase are actually lower as we show later.

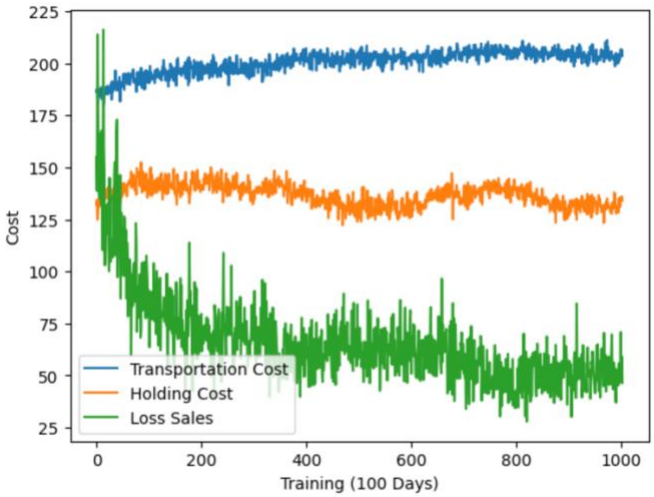


Figure 8. Distributions of Costs After for Improved Solution 2

After training the model, we simulated the system for 200 days. Below are the tabulated results we obtained for both our new solution and the older ones.

	Transportation Cost	Holding Cost	Lost sales	Total
Benchmark	-149.22	-236.94	-37.77	-423.93
Improved Solution 1	-181.46	-201.67	-7.71	-390.84
Improved Solution 2	-171.21	-119.36	-42.28	-332.85

Table 4. Comparison of Benchmark and Improved Solutions

As can be seen from the above results, our model was finally able to find a balance between the holding cost and lost sales while also keeping the transportation cost rather low. As our final model, we will stick with the same structure and grouping as we used in this solution and optimize the parameters of minimum and maximum order up to ratios that we used to replenishments. In this model we used 0.9 as ordering decision ratio and 1.1 and 2.2 as order up to ratio. We will try to optimize these parameters in our final model.

Final Model

We will keep the same structure and grouping we used in Improved Solution 2 for our final model. We will try to optimize the parameters of:

- Maximum $\frac{Inventory\ Level}{Order{-}Up{-}To\ Level}$ a store can have to be replenished (We were using 0.9 in the previous solution).

- The *ratio* in up to what order up to levels the stores will be replenished $ratio \times order - up - to\ level$ (We were using 1.1 and 2×1.1 in the previous solution).

The strategy we will follow will be based on basic trial and error. We will try increasing and decreasing the ratios and observe how model perform. Then at the end, we will take the best.

Results

We observed no improvement to the Improved Solution 2 after increasing and decreasing the ratios. We tried order up to ratios as 1.00, 1.05, and 1.2 and observed no improvement but a slight worsening effect. Increasing and decreasing the maximum $\frac{Inventory\ Level}{Order{-}Up{-}To\ Level}$ ratio caused no improvement too, resulting in us accepting the Improved Solution 2 as our final optimized model.

Conclusion

Table 5 summarizes the results of the final and previous models. After 200 days of simulation, final model was able to achieve an average reward of -332. Compared to the benchmark and improved solution 1, our final model was able to find the balance between the lost sales and holding cost while keeping the transportation cost low. We can say that our final model effectively learned when and how much to replenish especially for the stores that are far away from the warehouse and from other stores.

	Transportation Cost	Holding Cost	Lost sales	Total
Benchmark	-149.22	-236.94	-37.77	-423.93
Improved Solution 1	-181.46	-201.67	-7.71	-390.84
Final Model	-171.21	-119.36	-42.28	-332.85

Table 5. Comparison of the Models

References

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

https://towardsdatascience.com/learning-reinforcement-learning-reinforce-with-pytorch-5e8ad7fc7da0