Subjective Logic-based Reinforcement Learning for Optimization of Inventory Management

Name: Hyungmin Kim

***Index Terms*—dqn, uncertainty-aware reinforcement learning, dirichlet pdf, inventory management**

I. Introduction

Reinforcement Learning (RL) has emerged as a powerful machine learning paradigm worldwide, demonstrating as an effective method across diverse applications, including dynamic decision-making and autonomous systems. In inventory management, RL has proven particularly valuable for optimizing stock levels and minimizing operational costs.

In this project, I focused on applying RL to Inventory Management Optimization, leveraging stochastic modeling and uncertainty quantification to enhance decision-making. First I used foundational RL models to create environments for agents without uncertainty factors, then extended them through Dirichlet probability density functions (PDFs) and transformed into multinomial opinions. This integration aims to incorporate uncertainty metrics, such as vacuity, dissonance, and entropy, into the optimization process, enabling more robust handling of partially observable environments.

The core of my implementation involves customizing RL algorithms within a simulated retail environment. Using historical sales data from Kaggle [1], I modeled the interaction between sales, pricing, stock levels, and ordering policies. The environment tracks performances such as revenue, stockouts, and overstock penalties to ensure better options for decision-making.

A. Problem Statement

The three main objectives of this project is as follows:

Modeling the Sequential Problem: Formulate inventory management optimization method as a sequential Markov Decision Process (MDP), where actions involve ordering decisions, states capture stock levels and demand information, and rewards reflect the trade-off between holding costs and unmet demand penalties.

Developing RL Algorithms: Implement Deep Q-Networks (DQN), to optimize inventory policies in both deterministic and stochastic environments.

Evaluating Performance: Compare different RL-based policies and models, focusing on cost-efficiency and adaptability to uncertainty.

B. System Objective

1) Objective Function: The goal of this project is to **minimize the cumulative inventory management cost** over a predefined time horizon.

The objective function is defined as follows:

Objective Function:

$$\max_{\pi} \Big[R - (C_h + C_s + P_u) \Big]$$

- R: Total revenue generated from sales (Revenue).
- C_h: Holding costs incurred from maintaining inventory (Holding Cost).
- C_s: Penalty for stockouts, representing lost sales opportunities (Stockout Penalty).
- P_u: Penalty related to handling uncertainty in demand forecasting (Uncertainty Penalty).
- π: Policy used by the agent to optimize decisions.

Ultimately, the project aims to develop a scalable and robust RL-based inventory management solution that can handle various situations, providing real-time insights and adaptive decision-making capabilities.

C. Creativity of the Problem

Inventory management is a complex and dynamic challenge faced by lots of retail companies, as it involves making sequential decisions about replenishment and stock levels while balancing costs and service levels. [2] Traditional inventory optimization methods often assume deterministic demand and fixed lead times, which can not fully include sequential decisions. Real-world supply chains are ran with significant uncertainty, including fluctuating customer demand and changing holding or shortage costs. These complexities make it difficult for static models to adapt effectively to dynamic environments.

This project aim to use RL to offer a data-driven approach to address this problem by modeling inventory management as a sequential decision-making problem under uncertainty. Considering RL's characteristics that enables adaptive learning from interactions with the environment, making it suitable for managing unpredictable scenarios. By learning optimal replenishment policies, the agents can make decisions that minimize overall costs.

D. Preprocessing

1) Data Description: The dataset used in this project is the Retail Sales Data from Kaggle, collected from a Turkish retail company. The time period spans from the beginning of 2017 to the end of 2019.

The dataset consists of the following columns:

- **store_id**: The unique identifier of a store.
- **product_id**: The unique identifier of a product.
- **date**: The sales date in YYYY-MM-DD format.
- **sales**: The quantity of sales on that day.
- **revenue**: The total sales revenue for the day.
- **stock**: The stock quantity at the end of the day.
- **price**: The sales price of the product.
- **promo_type_1**: The type of promotion applied on channel 1.
- **promo_bin_1**: The binned promotion rate for the applied promo_type_1.
- **promo_type_2**: The type of promotion applied on channel 2.
- **promo_bin_2**: The binned promotion rate for the applied promo_type_2.
- **promo_discount_2**: The discount rate for applied promo type 2.
- **promo_discount_type_2**: The type of discount applied.

The shape of the dataset is as follows:

$$\text{Data Shape: } (19, 454, 838, 13)$$

This means the dataset contains 19,454,838 rows and 13 columns. The dataset contains 699 unique product IDs.

In order to use the dataset for reinforcement learning, it was necessary to simplify the data by excluding unnecessary columns. Only the minimum set of columns required for the clear objective of maximizing revenue were retained.

The columns that were dropped include:

- `store_id`
- `promo_type_1`
- `promo_bin_1`
- `promo_type_2`
- `promo_bin_2`
- `promo_discount_2`
- `promo_discount_type_2`

By dropping unnecessary columns, this step helps reduce the dimensionality of the data and retains only the relevant features needed for the model.

2) Handling Missing Value: The number of missing values is shown in the following image. Upon analyzing the missing values, it was observed that the missing data occurred specifically in the tail data from October 2019 to December 2019. Furthermore, as you can see at the "Fig. 1.", the proportion of missing values in the dataset ranged from approximately 3% to 6%. Given the large size of the dataset, this percentage was considered acceptable. As a result, rather than imputing the missing values, it was decided to exclude the affected rows from the analysis to maintain the integrity of the data. Rather than attempting to impute the missing values, it was decided to exclude these rows from the dataset to ensure the quality of the analysis.

3) Aggregating Data: Since the dataset had sales data separated by store, the data was consolidated to track the number of units sold per product on a daily basis. The data

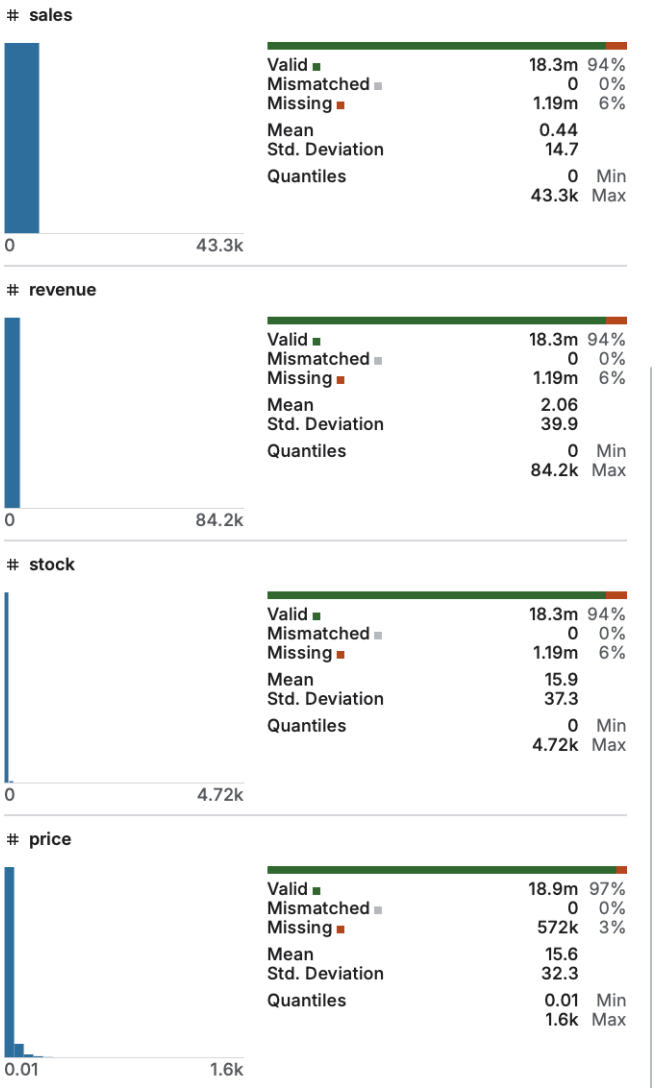


Fig. 1. Missing values count.

was then sorted in chronological order by day for the given time period.

4) Final Format and Splitting the Data into Training, Validation, and Test Sets: The dataset was divided into three subsets: training, validation, and test sets. This division was based on specific date ranges to simulate a real-world time-based prediction task, which is common in inventory management and sales forecasting.

- **Training Data (70%)**: The first portion, consisting of data from the start of the dataset until June 30, 2019, was used for training the model.
- **Validation Data (15%)**: The next portion, covering the period from July 1, 2019, to August 31, 2019, was used for tuning hyperparameters and finding the best model.
- **Test Data (15%)**: Finally, the test set, covering data from September 1, 2019, onward, was used to evaluate the final model's performance.

This splitting strategy ensures that the model is trained

on historical data and evaluated on future data, effectively preventing data leakage. The training data allows the model to learn the patterns, the validation data helps in optimizing the hyperparameters, and the test data gives the final performance assessment.

II. Reinforcement Learning Model

A. Definitions of States, Actions, and Rewards

1) State: The state is represented as a vector describing the stock level, sales, and price for each product on a given day. For each product, the state is expressed as a 3-dimensional array:

$$\text{State} = [\text{stock}, \text{sales}, \text{price}]$$

The environment updates this state dynamically based on the current date, and the Reinforcement Learning (RL) agent uses the state as input to decide the next action.

2) Action: The action corresponds to a vector representing the order quantities for each product. The action space is defined by the maximum allowable order quantity (`max_order`) for each product. The RL agent selects an order quantity for each product within the range:

$$0 \leq \text{order\_quantity} \leq \texttt{max\_order}$$

This action is represented as a vector with the same length as the number of products.

3) Reward: The reward function is carefully designed to enable the reinforcement learning agent to balance profitability and operational efficiency in inventory management. At the beginning of the project, I set up the reward function very simple. The components of the reward are:

$$\begin{aligned}\text{Reward} = (\text{Sales} \times \text{Price}) - \text{StockPenalty} \\ - \text{NoStockPenalty} - \text{StockMaintenanceCost}\end{aligned}$$

Each term in the reward function is defined as follows:

- **Sales Reward**: This is the revenue generated from sales, directly incentivizing the agent to meet customer demand:

$$\text{Sales Reward} = \text{sales} \times \text{price}$$

- **Stock Shortage Penalty**: A penalty is applied when stock falls below a critical threshold, preventing stockouts:

$$\text{NoStockPenalty} = \begin{cases} \text{no\_stock\_penalty} & \text{if new\_stock} \leq 30 \\ 0 & \text{otherwise} \end{cases}$$

- **Excessive Stock Penalty**: Overstocking is penalized to minimize unnecessary inventory holding:

$$\text{StockPenalty} = \begin{cases} \text{stock\_penalty} & \text{if new\_stock} > 0.8 \times \text{max\_stock} \\ 0 & \text{otherwise} \end{cases}$$

- **Inventory Holding Cost**: The cost of maintaining inventory is included as a linear penalty:

$$\text{StockMaintenanceCost} = \text{holding cost per unit} \times \text{new\_stock}$$

B. Deep Q-Network (DQN) Algorithm

In this project, I utilized Deep Q-Network (DQN) to address the inventory management problem. DQN extends the classical Q-Learning algorithm, which is a model-free reinforcement learning approach. Q-Learning works by iteratively updating a Q-value table that represents the expected utility of actions taken in different states. However, I first tried to fit model with multiple products, which might cause high level of state-action space. For this specific problem, after taking a consideration, DQN was selected as the most suitable reinforcement learning algorithm for this project as you can see at the "TABLE 1":

- **Discrete Action Space**: The inventory management problem involves a discrete action space, where actions are limited to specific inventory quantities. DQN is particularly effective for such environments because it maps states to actions in a way that maximizes expected rewards.
- **Scalability**: The use of neural networks for approximating the Q-function allows DQN to scale efficiently, handling larger state spaces, which is crucial for managing numerous products and stores.
- **Environment Simplicity**: The environment has relatively low continuous complexity, with fewer state variables, making DQN's Q-value approximation effective and computationally feasible.

I designed the following neural network architecture as the Q-network:

```
def _build_model(self):
    model = nn.Sequential(
        nn.Linear(self.state_size, 128),
        nn.ReLU(),
        nn.Linear(128, 128),
        nn.ReLU(),
        nn.Linear(128, self.action_size),
        nn.Sigmoid()
    )
    return model
```

This architecture consists of:

1) **Input Layer**: The input dimension (`state_size`) corresponds to the number of state variables as I mentioned before: [stock, sales, price].
2) **Hidden Layers**: Two fully connected layers with 128 neurons each, activated by the ReLU function, allowing the network to capture non-linear relationships.
3) **Output Layer**: A fully connected layer with a size equal to the action space (`action_size`), which includes [max_order + 1] numbers of actions(amount of orders, 0 max_order) After that, I used a sigmoid activation function to produce outputs in the range $[0, 1]$, suitable for determining inventory replenishment actions. This allowed the q-value as a positive value so as to ensure the alpha value is not in the negative.

The DQN algorithm works as follows:

TABLE I
COMPARISON OF REINFORCEMENT LEARNING ALGORITHMS

Algorithm	Characteristics
Q-Learning	Simple and easy to implement; Suitable for small, discrete state/action spaces; Inefficient in large state/action spaces; Not directly applicable to continuous states (requires function approximation).
Deep Q-Networks (DQN)	Extends Q-Learning with neural networks; Can learn in large state spaces; Efficiency decreases with large action spaces; Doesn't account for the continuity of the environment; Separates policy evaluation and improvement.
Proximal Policy Optimization (PPO)	Stable learning process; Suitable for continuous state/action spaces; Computationally efficient; Requires hyperparameter tuning; Relatively complex to implement.
Policy Gradient (PG)	Handles both continuous and discrete action spaces; Directly learns the optimal policy; High variance leads to lower learning stability; Less efficient.
Actor-Critic	Combines advantages of Policy Gradient and Value-based RL; Simultaneously performs policy learning and value estimation; Sample efficient; Relatively complex to implement; Sensitive to hyperparameter settings.

Algorithm 1 DQN Algorithm

Initialize Q-network $Q(s, a; \theta)$ and target network $Q(s, a; \theta^-)$ with random weights
Initialize experience replay memory D
for each episode **do**
  Initialize state s_0
  for each step in the episode **do**
    Select action a_t using epsilon-greedy policy:

$$a_t = \begin{cases} \text{random action}, & \text{with probability } \epsilon \\ \arg\max_a Q(s_t, a; \theta), & \text{otherwise} \end{cases}$$

    Execute action a_t, observe reward r_t and next state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in D
    Sample a mini-batch of transitions (s_j, a_j, r_j, s_{j+1}) from D
    Compute the target:

$$y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta^-)$$

    Perform a gradient descent step on:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_j (y_j - Q(s_j, a_j; \theta))^2$$

    Every C steps, update target network: $\theta^- \leftarrow \theta$
  end for
end for=0

In my project, the Q-network is constructed in _build_model and initialized with random weights using PyTorch, with actions selected through an epsilon-greedy policy in the act method. Experience replay is implemented using a deque, where transitions are stored and sampled for training in the replay method. The target for updates follows the DQN formula, $r + \gamma \max Q(s_{t+1}, a'; \theta)$, and the loss is calculated using `nn.MSELoss` with backpropagation performed via the Adam optimizer. However, the implementation currently lacks a separate target network θ^-, which is essential for stabilizing training in the DQN algorithm. Additionally, uncertainty measures such as vacuity, dissonance, and entropy are incorporated to adjust rewards dynamically, introducing a novel element beyond the standard DQN framework. Training is orchestrated in the train_and_validate function, which iteratively updates the Q-network through experience replay.

C. Hyperparameter Settings

1) Learning Rate: The learning rate (η) determines the step size for weight updates during training. It directly impacts the convergence speed and stability of the learning process. I first initialized it as (0.001

2) Gamma (Discount Factor): The discount factor (γ) controls the agent's emphasis on immediate versus future rewards. Initial value was high values (e.g., 0.99) which bias the agent toward considering long-term rewards.

3) Epsilon Decay: Epsilon controls the balance between exploration and exploitation. The epsilon value starts high (for exploration) and decays over time, encouraging the agent to exploit the learned policy as training progresses.

4) Batch Size: The batch size in experience replay is set to 32, meaning that the agent samples 32 experiences from the replay buffer to update the model at each training step.

5) Replay Buffer Size: The size of the replay buffer affects how many past experiences the agent can store for training. A buffer of 2000 experiences provided a balance between capturing diverse samples and maintaining computational feasibility.

6) Maximum Stock and Order Limits:

- **Maximum Stock** (1000): Defines the upper limit of stock for each product.
- **Maximum Order** (30): Restricts the agent's action space to feasible ordering quantities.

These constraints reflect real-world limitations and influence the agent's policy learning.

D. Environmental Dynamics and Its Influence on Decision-Making

1) Uncertainty: The environment includes an uncertainty factor, represented by the entropy of the product's stock, sales, and price. High uncertainty leads to a penalty, encouraging the agent to make more confident and informed decisions.

2) Stock Dynamics: Stock levels change dynamically based on sales and orders. The agent must learn to manage stock levels effectively by ordering the right amount of products to avoid shortages or excess stock. Penalties are applied for both stock shortages and overstocking, making it crucial for the agent to find an optimal ordering strategy.

3) State Updates: The state is updated at each time step based on the current stock, sales, and price. The agent uses this updated state to choose the next action, allowing it to adapt to changing conditions and optimize its policy over time.

E. Model Optimization and Experimentation

Initially, the agent was designed to determine the order quantities for all products in the dataset simultaneously. However, with around 700 products and thousands of episodes in the training process, this approach proved to be highly computationally expensive and time-consuming. As a result, the training process faced significant delays, which hindered the progress of experimentation.

To mitigate this issue, I decided to simplify the problem by focusing on a single product for the initial experiments. This approach allowed for more manageable training times while still retaining the essential dynamics of the inventory management problem. The model was thus adapted to perform actions for a single product at a time, significantly improving the efficiency of the learning process.

Following this modification, I conducted a series of experiments using the base reinforcement learning model to test its performance. The results from the initial model, denoted as Model 1, were as follows:

- **Test Reward of Model 1** (lr = 0.001, ϵ decay = 0.995, γ = 0.99): 410.50
- **Model 1 with Validation Reward**: 2523.25
- **Test Reward of Model 1**: 369.66

I conducted These test results demonstrated the agent's ability to learn a basic inventory management policy using the base RL model. The large difference between the test reward and validation reward indicated that the model may have overfitted to the training data, which is a common issue in reinforcement learning. Nevertheless, this experiment helped me to set a baseline for the project and experiment fine tuning.

III. Uncertainty-Introduced Environment and Quantification

A. Converting Into a Dirichlet PDF

The action selection policy in the `DQNAgent` leverages the Dirichlet Probability Density Function (Dirichlet PDF) to model the uncertainty in Q-values for various actions. Here, I set the q_values using the formula below:

```
q_values = softmax(q_values)
alpha = q_values * 10

dirichlet_sample
= dirichlet.rvs(alpha, size=1).flatten()
```

The following steps describe the mechanism and purpose of the integration:

1) **Softmax Transformation:** The Q-values predicted by the policy are transformed into a probability distribution using the softmax function:

$$q_i = \frac{e^{q_i}}{\sum_j e^{q_j}}$$

This ensures that the Q-values are normalized and sum to 1, representing valid probabilities.

2) **Concentration Parameter:** A concentration parameter (α) for the Dirichlet distribution is computed by scaling the probabilities:

$$\alpha_i = q_i \cdot \text{scale factor}$$

Larger α values indicate greater certainty in the model's predictions, whereas smaller values introduce more uncertainty.

3) **Sampling from the Dirichlet Distribution:** A sample is drawn from the Dirichlet distribution, which represents the updated belief over the action probabilities:

$$\text{dirichlet\_sample} \sim \text{Dirichlet}(\alpha)$$

This sample encapsulates both the predicted likelihoods of actions and their associated uncertainty.

The Dirichlet PDF serves as a probabilistic model for the action probabilities. By incorporating both the predicted likelihood and the uncertainty of the Q-values, the approach enables more informed and adaptive decision-making within the agent.

B. Transformation of Dirichlet PDF into a Multinomial Opinion

Then I converted the Dirichlet sample into a multinomial opinion, which serves as a normalized representation of belief over possible actions.

```
def compute_multinomial_opinion(dirichlet_sample):
    opinion
    = dirichlet_sample / np.sum(dirichlet_sample)
    return opinion

opinion
= compute_multinomial_opinion(dirichlet_sample)
```

Through this code, I normalized dirichlet_sample to form a probability distribution and got multinomial opinion.

$$\text{opinion}_i = \frac{\text{dirichlet\_sample}_i}{\sum_j \text{dirichlet\_sample}_j}$$

This normalized opinion reflects the relative confidence in each action, derived from the updated belief state represented by the Dirichlet sample.

Multinomial opinions provide a structure of the system's belief in action probabilities. This facilitates more informed decision-making under uncertainty, aligning the agent's behavior with its belief structure.

The model gives three types of uncertainty metrics, each highlighting a unique aspect of uncertainty in the predictions.

1) Vacuity: Represents the lack of evidence, calculated as:

$$\text{vacuity} = 1 - \frac{\sum_i \text{dirichlet\_sample}_i}{\text{len(dirichlet\_sample)}}$$

Higher vacuity indicates insufficient information to strongly support any particular action.

2) Dissonance: Represents conflict in the belief state, computed as:

$$\text{dissonance} = \max(\text{dirichlet\_sample}) - \min(\text{dirichlet\_sample})$$

Larger dissonance suggests greater disagreement within the predicted probabilities.

3) Entropy: Measures the overall spread or uncertainty in the distribution, defined as:

$$\text{entropy} = -\sum_i \text{dirichlet\_sample}_i \cdot \log(\text{dirichlet\_sample}_i)$$

Higher entropy reflects a more uncertain or dispersed belief over actions.

These metrics provide a comprehensive view of uncertainty by quantifying:

- **Vacuity:** The extent to which the model lacks evidence to support any particular action.
- **Dissonance:** The internal conflict within the predicted probabilities.
- **Entropy:** The overall spread or randomness in the belief state.

By integrating these metrics, the agent gains deeper insights into its confidence in predictions. This information guides the reward mechanism and improves the decision-making process by accounting for various aspects of uncertainty.

So at this point, I added these metrics into the reward function so that the agent can take the uncertainty as a consideration for rewards. Finally, the confirmed RL reward is as follows:

$$\begin{aligned}\text{Reward} = (\text{Sales} \times \text{Price}) + \text{OrderReward} - \text{StockPenalty} \\ - \text{NoStockPenalty} - \text{StockMaintenanceCost} \\ - \text{UncertaintyPenalty}\end{aligned}$$

```
# 1. The sum of the values in the opinion
rewards -= np.sum(opinion) * 5

# 2. Sales
rewards += sales * price

# 3. Optimal order quantity
if abs(order_quantity - sales)
<= self.max_order * 0.1: rewards += 20

# 4. Out of stock
if new_stock <= 30:
    rewards -= self.no_stock_penalty

# 5. Overstock
elif new_stock > self.max_stock * 0.8:
    rewards -= self.stock_penalty

# 6. Inventory holding cost
rewards -= new_stock * 0.01

# 7. Uncertainty
if vacuity > 0.5:
    rewards -= 10
if dissonance > 0.4:
    rewards -= 10
if entropy_value > 0.7:
    rewards -= 10
```

The reward function is composed of 7 ways of penalty and rewards.

4) Model Optimization and Experimentation: After converting into multinomial opinion from dirichlet PDF, I set four differences in hyperparameters:

The conditions other than the hyperparameters were applied the same as the base RL, and the model was trained for a total of 3000 episodes. As you can see at the "Fig 2", among the four models, Model 4 showed the best performance with a Validation Reward of 2690.65, and when tested on the Test data, it achieved a Reward of 2708.0. The results of this test show that Model 4 had a relatively moderate learning rate, higher epsilon decay value, and also a higher gamma compared to the other models. In other words, Model 4 converged adequately and completed exploration and focused on exploitation faster than the other models, and it concentrated on long-term rewards. Therefore, in a scenario where all the models were overfitting, Model 4, which converged the most appropriate time and finished exploration earlier can be interpreted as performing the best.

TABLE II
HYPERPARAMETERS TABLE

Model	Learning Rate	Epsilon Decay	Gamma
1	0.001	0.995	0.99
2	0.005	0.998	0.95
3	0.0005	0.990	0.90
4	0.002	0.997	0.98

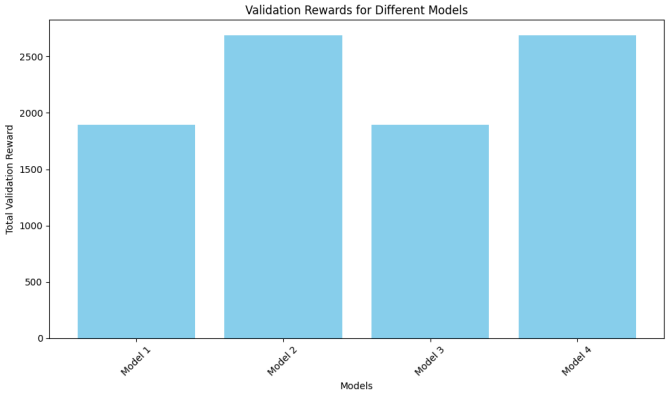


Fig. 2. Validation rewards of slrlModels1

"Fig. 3" shows the reward during training using the training data over the course of the episodes. After approximately 300 episodes, the reward converges to a specific value and no longer shows significant movement. This indicates that the model has overfitted, and adjustments were made in the next training session to address this issue. Additionally, the number of layers in the neural network was reduced, and dropout was applied to prevent excessive overfitting. In "Fig. 4," it can be observed that the loss function did not decrease and showed large fluctuations. To resolve this, the batch size was reduced to stabilize the training.

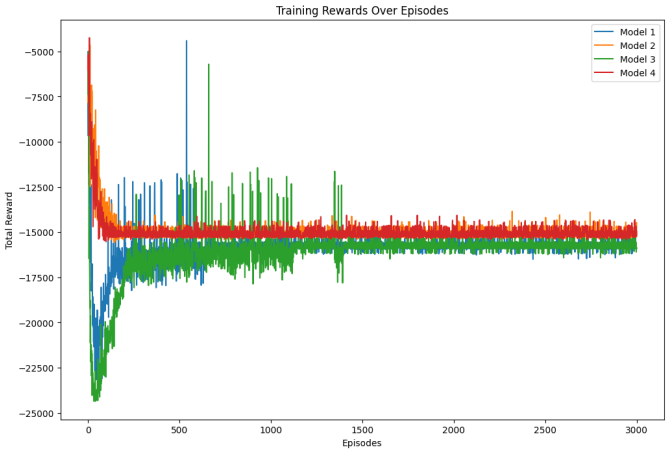


Fig. 3. Training reward of slrlModels1

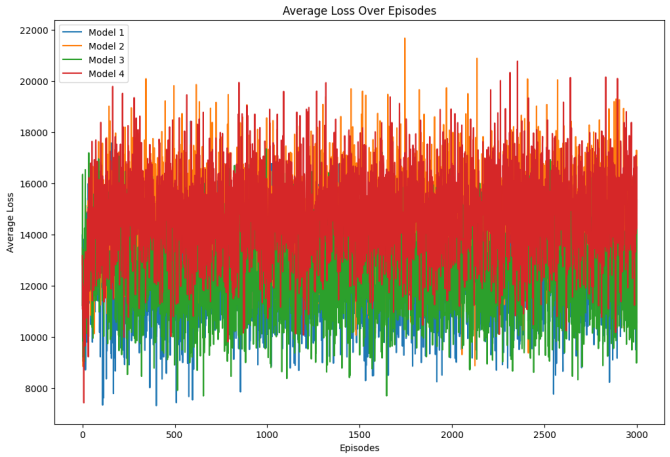


Fig. 4. Loss function of slrlModels1

C. Effectiveness and Efficiency

Based on the improvements from the previous model, the training data was used to train a new model. In the final model, the number of episodes was reduced to 500, the number of neurons in the neural network was decreased from 128 to 64, and dropout was applied to address overfitting. Additionally, the batch size was reduced from 32 to 16, which helped achieve a more stable loss function.

```
def _build_model(self):
    model = nn.Sequential(
        nn.Linear(self.state_size, 64),
        nn.ReLU(),
        nn.Dropout(p=0.5),
        nn.Linear(64, 64),
        nn.ReLU(),
        nn.Dropout(p=0.5),
        nn.Linear(64, self.action_size),
        nn.Sigmoid()
    )
    return model
```

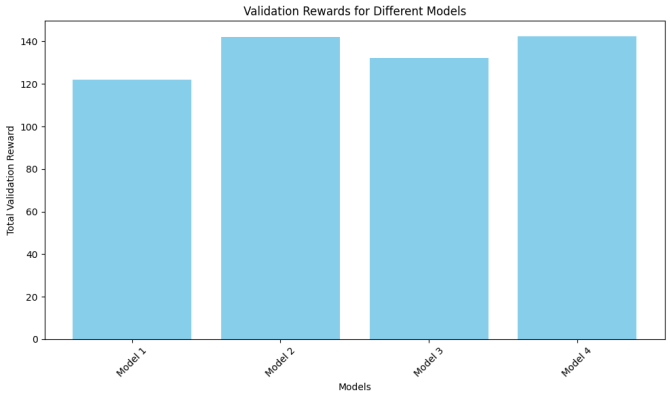


Fig. 5. Performance of slrlModels2

The models in this experiment all showed rapid convergence without overfitting. Additionally, Model 4 achieved the highest validation reward of 142.47, indicating it was the best-performing model. One interesting observation is that, similar to previous experiments, Model 4 performed the best. [3] This suggests that an appropriate learning rate in the middle range, combined with a quick exploration phase followed by long-term data learning, positively influences model performance.

```
Effectiveness (Total Reward on Validation): 102.11995937347413
Efficiency (Mean Loss during Training): 1384.9000055232902
```

Fig. 6. Effectiveness and Efficiency of slrlModels2

Finally, the Effectiveness of the model was measured by the Total Reward on Validation, which resulted in a value of 102.120. Additionally, Efficiency was measured by the Mean Loss during Training, yielding a value of 1384.90.

TABLE III
MODEL VALIDATION REWARDS

Model	Validation Reward
Model 1	122.11
Model 2	142.15
Model 3	132.20
Model 4	142.47

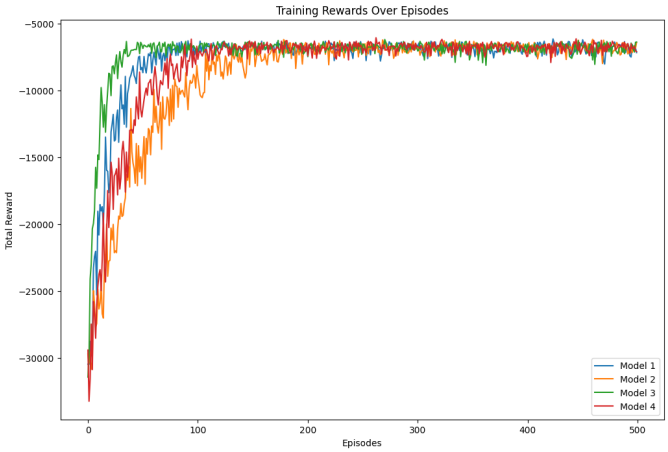


Fig. 7. Training reward of slrlModels2

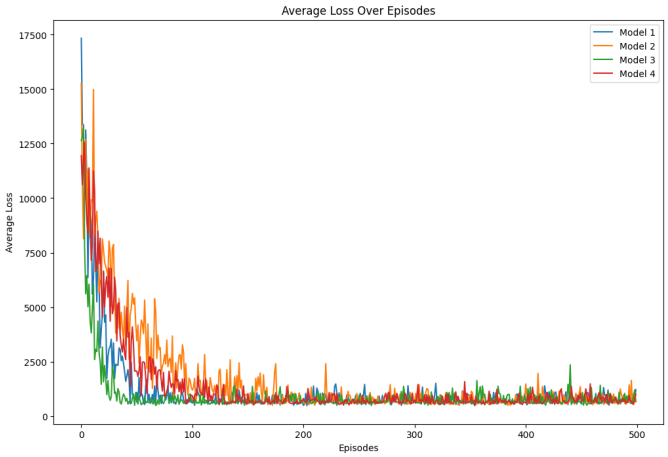


Fig. 8. Loss function of slrlModels2

D. Evaluation of Uncertainties

During each step in the environment of the dqn class's step function, the Dirichlet sample is drawn and used to calculate the three uncertainty metrics: vacuity, dissonance, and entropy. These metrics are stored in for each of the metrics across episodes. After training, the uncertainties are plotted over the episodes, showing the evolution of vacuity, dissonance, and entropy throughout the training process.

```
Effectiveness (Total Reward on Validation): 102.11995937347413
Efficiency (Mean Loss during Training): 1384.9000055232902
```

Fig. 9. Effectiveness and Efficiency of slrlModels2

The Effectiveness of the model was measured by the Total Reward on Validation, which resulted in a value of 102.120. Additionally, Efficiency was measured by the Mean Loss during Training, yielding a value of 1384.90.

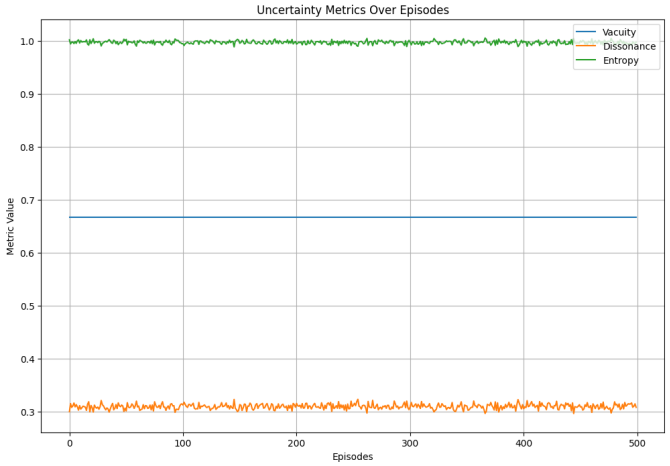


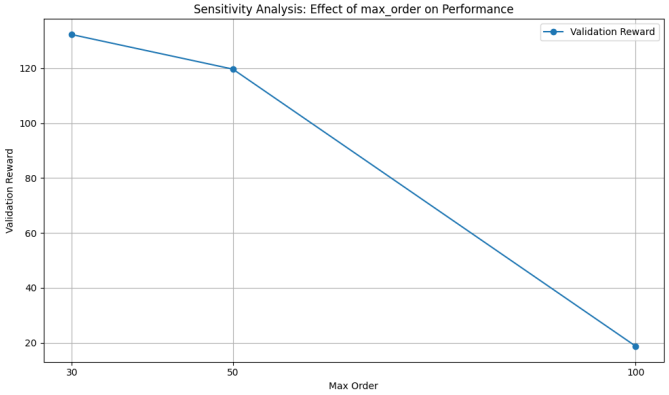
Fig. 10. Uncertainties of slrlModels2

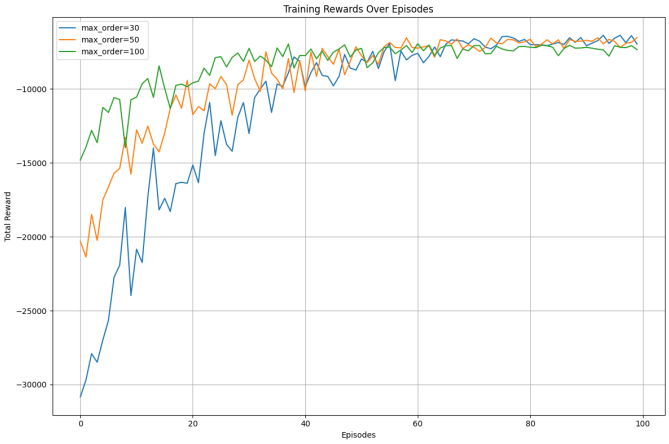
As shown in "Fig. 9", vacuity remains almost constant at around 0.67, while entropy and dissonance exhibit subtle fluctuations around 1.0 and 0.3, respectively. This suggests that the model is struggling to reduce uncertainty. A vacuity value of 0.67 indicates that the model consistently holds more than half of the uncertainty, implying low confidence in its predictions. Additionally, the entropy being close to 1 reflects high uncertainty, indicating that the model experiences considerable confusion in its predictions. Another possibility is that the model has not yet been fully optimized and is still experimenting with various actions during its learning process. Regarding dissonance, the model shows relatively balanced confidence across its predictions, though some slight imbalance remains, which should be noted.

E. Sensitivity Analysis

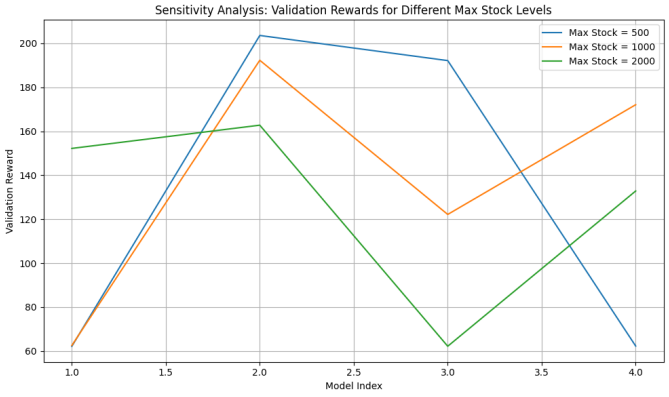
A comprehensive sensitivity analysis was performed to assess the impact of varying key parameters on the agent's performance. I analyzed two primary parameters: max_order and max_stock.

In the first analysis, the value of max_order was adjusted to [30, 50, 100]. As illustrated in Fig, the model achieved the best performance when max_order was set to 30. This result suggests that increasing the maximum allowable order quantity leads to suboptimal outcomes due to excessive overstock and the associated increase in holding costs. The penalties incurred from these inefficiencies outweigh any potential benefits of placing larger orders, thereby negatively impacting the agent's overall performance.





The second analysis examined the influence of max_stock with values set at [500, 1000, 2000]. The max_stock parameter was tested across all four models used in the last training session, and an intriguing observation was made. Depending on the model, high max_stock values correlated with better performance in some cases, while low max_stock values resulted in better performance in others. If only a single model had been used, just deciding the optimal max_stock value might have been more straightforward. However, testing across multiple models revealed the need for further training to draw definitive conclusions about the relationship. For example, in the case of Model 1, the best performance was observed when max_stock was set to 2000. In contrast, Models 2 and 3 showed



better performance with lower max_stock values. This discrepancy can be attributed to the influence of differing hyperparameters such as learning rates and epsilon on the max_stock parameter.

IV. Conclusion

Through this project, I implemented a DQN-based reinforcement learning model and integrated uncertainty measures into the reward function to improve decision-making accuracy. By utilizing experience replay and target networks, I enhanced the stability of the training process and experimented with various hyperparameters to achieve optimal performance.

In conclusion, incorporating uncertainty metrics into the reward function significantly impacted the model's decision-making, and the experiments confirmed the potential for effectively applying reinforcement learning in such contexts. Moving forward, I expect to extend this approach to more complex environments.

References

[1] https://www.kaggle.com/datasets/berkayalan/retail-sales-data/data

[2] Dehaybe, H., Catanzaro, D., Chevalier, P. (2024). Deep Reinforcement Learning for inventory optimization with non-stationary uncertain demand. European Journal of Operational Research, 314(2), 433–445. https://doi.org/10.1016/j.ejor.2023.10.007

[3] Ladosz, P., Weng, L., Kim, M., Oh, H. (2022). Exploration in deep reinforcement learning: A survey. Information Fusion, 85, 1–22. https://doi.org/10.1016/j.inffus.2022.03.003