Stocktake

Revised, December 2024

Overview

- Improving the model: Exploring the idea of implementing reinforcement learning methods (this will be on hold for now).
- Introducing more complex cases to the model: Ensuring the current model is robust.

To-do:

- Implement different policies (maybe do some research on this matter to see what inventory policies there are)
- Attempt to convert e2e.py from using tensorflow to pytorch
- Implement condition to allow inventory to fall below 0 for SS, fixed order and order-up-to policy
- Implement logic where if there's an existing outstanding order, we do not trigger another order. (WIP)

Checklist

- Implemented random seed function to ensure results are reproducible
- Accounted for edge cases where main.py breaks when the duration range is out of bound (i.e. index 60 is out of bounds for interval 60)
- Implemented date based time-series function instead of set intervals, enabling reorder interval to be set to certain days of week
- Implmented different distributions

```
# Before
for i in range(1, horizon):
    inventory_level[i] = inventory_level[i - 1] + inventory_level[i] - future_demand[i]
    if reorder_point_arr[i] and t <= len(po_reaching):
        vm = po_reaching[t]
        delta_t = int(np.floor(b * (po_reaching[t + 1] - po_reaching[t]) / (h + b)))  #
                                                                    s*
        cum_demand = np.sum(future_demand[vm:vm + delta_t])
        optimal_qty = max(cum_demand + np.sum(future_demand[i:vm]) - inventory_level[i]
                                                                , 0)
        inventory_level[vm] += optimal_qty
        t += 1
        print(optimal_qty)
```

```
# After
for i in range(1, horizon):
    # Deduct inventory based on demand
    inventory_level[i] = inventory_level[i - 1] + inventory_level[i] - future_demand[i]

    if reorder_point_arr[i] and t < len(po_reaching):  # Ensure `t` is in bounds
        vm = po_reaching[t]

        # Dynamically extend `inventory_level` if `vm` is out of bounds
        if vm >= len(inventory_level):
```

```
            # Extend the array to include the required index
            new_size = vm + 1  # Ensure it can accommodate `vm`
            inventory_level = np.append(inventory_level, np.zeros(new_size - len(
                                                    inventory_level)))

        delta_t = int(np.floor(b * (po_reaching[t + 1] - po_reaching[t]) / (h + b)))  #
                                                    Calculate `s*`

        # Dynamically extend `future_demand` if `vm + delta_t` is out of bounds
        while vm + delta_t >= len(future_demand):
            future_demand = np.append(future_demand, abs(get_demand(mu_demand,
                                                    std_demand, interval + 1)))

        cum_demand = np.sum(future_demand[vm:vm + delta_t])  # Calculate cumulative
                                                    demand over `delta_t`

        # Calculate the optimal reorder quantity
        optimal_qty = max(cum_demand + np.sum(future_demand[i:vm]) - inventory_level[i]
                                                    , 0)

        # Update inventory level at `vm`
        inventory_level[vm] += optimal_qty
        t += 1  # Move to the next reorder point

        print(f"Optimal quantity at time {i}: {optimal_qty}")  # Debug output
```

Iterations for main.py

- v2: to account for edge cases
- v3: Converted period output to date format
- v4: Converted interval from fixed defined value to day of week
- v5: Included function to output data into various .csv files
- v6: Account for different distributions (without labelling policy)
- v7: Account for different distributions (with labelling policy)
- v8: Similar to v7 but removed log-norm dist
- v9: Incomplete implementation of periodic review with constant order quantity policy
- v10: Incomplete implementation of periodic review with constant order quantity policy (another approach)
- v11: Implementation of periodic review with constant order quantity policy (Simplified, demand to follow sS policy but fixed restock quantity)
- v12: Implemented order-up-to policy.
- v13 to v15: Imeplementataion of all 4 policies with reorder point error and misc edits on plot.
- v16: Implemented policies based off specifications with reorder point constraint.
- v17: Updated .csv output format.

Various Distribution

Poisson Distribution

- **Use Case:** Data is discrete and represents count-based events
- **Parameters:** Single parameter, λ (mean demand rate)
- **Advantages:** Naturally models discrete
- **Disadvantages:** Assume variance equals mean

Exponential Distribution

- **Parameters:** Rate parameter $\lambda = \frac{1}{\mu}$ (inverse of mean)

Log-Normal Distribution

- **Use Case:** If demand or lead time is positively skewed (i.e. seasonal demand spikes or random disruptions in lead time)
- **Parameters:** μ and σ of underlying norm dist
- **Advantages:** Captures positively skewed data, always non-negative

Gamma Distribution

- **Use Case:** If lead time or demand is non-negative and exhibits skewness with extreme tail behavior (i.e. Supplier delays due to unpredictable factors like weather)
- **Parameters:** Shape (k) and scale (θ)
- **Advantages:** Flexible for modelling non-negative, skewed data

Bionomial Distribution

- **Use Case:** Demand is tied to a fixed number of trials (i.e. numbers of customers served, ecah having a probability of purchasing)
- **Parameters:** Number of trials (n) and success probability (p)
- **Advantages:** Handles scenarios where demand cannot exceed a fixed limit
- **Disadvantages:** Limited to bounded scenarios

Weibull Distribution

- **Use Case:** Modelling lead time in scenarios where failture times or delivery times exhitbit a wide range of variability (i.e. delivery times for items transported across different distances or routes)
- **Parameters:** Shape (k) and scale (λ)

Beta Distribution

- **Use Case:** If demand is bounded within a range but exhibits variability in certain subranges (i.e. more demand during weekends or holidays)
- **Parameters:** Shape (α, β)
- **Advantages:** Captures bunded, asymmetric distributions

Heavy-Tailed Distribution

- **Use Case:** If demand or lead time has extreme outliers that occur with a higher-than-normal freq
- **Parameters:** Shape and scale parameters
- **Advantages:** Account for black swan events

Various Policy

Newsvendor Policy

- Optimizes inventory to balance holding costs (overstocking) and stockout costs (understocking)
- **Features:** Optimal order quantity is based on demand distribution and service level targets.

Economic Order Quantity Policy

- Focus on determining the optimal order quantity that minimizes total inventory costs (holding costs and ordering costs)
- **Use Case:** Suited for stable demand environment with predictable lead times
- **Features:** Fixed order quantity, orders are placed when inventory levels hit the reorder point
- **Limitations:** Does not adapt dynamically to demand fluctuations

Base Stock Policy

- Inventory is replenished to a predetermined base stock level after each demand
- **Features:** Ideal for high-value or low-demand items, used often in make-to-order or perishable goods scenarios (i.e. healthcare industry where stockouts are costly)
- **Limitations:** High administrative effort since orders happen frequently

s,Q Policy

- Combines a fixed order quantity (Q) with a reorder point (s). When the inventory falls below s, a replenishment order size Q is placed.
- **Features** Orders are consistent in size, inventory is reviewed continuously (Moderate variability in demand or lead time)
- **Limitations:** Requires continuous monitoring

Periodic Review Policy (R, S)

- Inventory levels are reviewed at fixed intervals (R), and the inventory is replnished up to a target level S. (Use when regular reviews are more feasible than continuous monitoring)
- **Features:** Orders placed periodically
- **Limitations:** Higher probability of stockouts if review intervals are too long

Continuous Review (Q, r) Policy

- A replenishment order of size (Q) is placed whenever the inventory position reaches the reorder point (r)
- **Use Case:** Requires continuous tracking systems
- **Features:** Monitors inventory continuously, Q is often calculated based on EOQ principles

Min-Max Policy

- Inventory is replenished when it falls below a minimum level (min) and is restocked up to the maximum level (max)
- **Features:** Doesn't require precise demand forecasting
- **Limitations:** Inefficient for high-value items or highly variable demand

invutils.py

get_safety_stock

Buffer stock to maintain the desired service level after taking into consideration variability in both lead time and demand.

$$\text{Safety Stock} = Z\sqrt{\text{Mean Lead Time} * \text{Demand Variance} + \text{Demand Mean}^2 * \text{Variance Lead Time}}$$

Parameters

- **lead_time:** A list of lead times
- **demand:** A list of demand values
- **service_level:** The desired service level

Variables

- **z_score:** The number of s.d required to achieve the specificed service level under a normal distribution. Using st.norm.ppf, we computes the inverse of the cumulative distribution function
- **lead_time_mean:** Computes the mean lead time
- **lead_time_var:** Computes the variance of lead
- **demand_mean:** Computes the mean demand
- **demand_var:** Computes the variance of the demand

get_reorder_point

Parameters

- **Horizon:** Number of days in the review period (Number of days in the future for optimal inventory position)
- **Interval:** Number of days between consecutive reorder poitns

1. **Initialize an Array (rp)** rp = np.zeros(horizon, dtype=np.int8)
   - Creates a NumPy array of size horizon filled with zeros, where each position represents a time step.
   - Since the values in the array will only be 0 or 1, the data type is int8 to save memory.
2. **Set Reorder Points** cnt = 2
   - Starts from index 2 in the array (0-based indexing), meaning the first reorder point is set after skipping the first two time steps.
   - Inside the while loop:
     - rp[cnt] = 1: Marks the position at cnt as a reorder point by setting it to 1.
     - cnt += interval: Moves the counter forward by the specified interval.

get_target_inventory

get_lead_time

Generates a list of lead times basede on a normal distribution.

- **Generate Random Numbers:** np.random.normal(loc=mean, scale=std, size=n) generates n random values from a normal distribution with:
  - Mean (loc) = mean.

  - Standard deviation (scale) = std.

- **Ensure Non-Negative Values:** The np.abs function takes the absolute value of each lead time. Since negative lead times don't make sense in real-world scenarios, this step ensures all values are positive.
- **Round the Values:** Using np.round, the generated lead times are rounded to the nearest integer. This is appropriate if lead times are measured in whole units.
- **Return the Lead Times:** The function returns a NumPy array of rounded, non-negative lead times.

get_demand

Geenrates demand values based on a normal distribution.

- **Generate Random Demand Values:** np.random.normal(loc=mean, scale=std, size=n) generates n random values from a normal distribution with the specified mean (loc = mean) and standard deviation (scale = std).
- **Ensure Non-Negative Values:** The np.abs function ensures all demand values are non-negative by converting negative values (which don't make sense for demand) into positive values.
- **Round the Values:** np.round(...) rounds the generated demand values to the nearest integer. This is appropriate if demand is measured in whole units/
- **Return the Simulated Demand:** The function returns a NumPy array of rounded, non-negative demand values.

main.py

Simulates and visualizes inventory management over a fixed period using a combination of inventory replenishment conditions.

Objectives

1. Simulate inventory dynamics over a 60 day horizon under uncertainty in demand and lead time.
2. Implement and compare two policies for inventory replenishment:
   - **sS Policy:** Orders are placed to bring inventory up to a certain level when it drops below a threshold.
   - **Labelling Policy:** A more advanced policy that dynamically adjusts reorder quantities based on demand forecasts.
3. Visualize inventory levels, reorder points, and order arrivals.

Parameters

1. **Initial Conditions:**
   - initial_inventory = 100: Initial stock at the start of the simulation.
   - horizon = 60: Simulates inventory over 60 days.
   - interval = 10: Fixed interval between reorder points (every 10 days).
2. **Demand and Lead Time:**
   - mu_demand = 10: Mean for daily demand
   - std_demand = 6 : Standard deviation for daily demand
   - mu_lead_time = 5 Mean for order lead times
   - std_lead_time = 1: Standard deviation for order lead times
3. **Service Level:**
   - service_level = 0.99: Ensures sufficient stock to meet demand 99% of the time.

Simulation Process

1. **Generate Reorder Points**
   - The function get_reorder_point(horizon, interval) creates a binary array marking reorder days.
2. **Generate Historical and Future Data**
   - history_lead_time and history_demand are simulated to mimic historical patterns of lead times and demand.
   - future_lead_time and future_demand are generated for the simulation horizon and upcoming reorders.
3. **Calculate Safety Stock**
   - The get_safety_stock function computes a buffer stock to account for demand and lead time variability.

sS Policy (Inventory Simulation)

1. **Inventory Tracking:**
   - inventory_level tracks daily inventory levels.
   - At each time step (i), inventory is adjusted for demand (future_demand[i]).
2. **Reorder Logic:**
   - On a reorder day (reorder_point_arr[i]), reorder quantity is computed based on:
     - Criterion 1: Sufficient stock to cover demand during lead time.
     - Criterion 2: Sufficient stock until the next reorder point.
   - If either criterion is violated, a reorder is triggered. The order quantity ensures the inventory level satisfies the respective thresholds.
3. **Order Arrival:**
   - Orders placed arrive after their corresponding lead time (future_lead_time[reorder_ptr]) and are added to the inventory.

Labelling Policy

- **Simulates an alternative policy:** Reorder quantities are calculated based on the cumulative demand between current and next reorder points, adjusted dynamically.

e2e.py

Implements e2e for inventory management and forecasting.

Main Functionality

1. **Simulates inventory replenishment data** for multiple stores and SKUs over a horizon of 60 days.
2. **Trains a DCNN model** to predict optimal reorder quantities based on dynamic features such as inventory levels, demand, and lead time.

Detailed Explanation

1. Data Generation

Generates synthetic training data for the inventory replenishment process.

SKU and Store Loops

The script loops through MAX_SKU_ID (100) and MAX_STORE_ID (50) to simulate the behavior of different products and stores.

Inventory Dynamics

- Each SKU-store combination is initialized with random parameters such as initial_inventory, mu_demand, and mu_lead_time.
- Replenishment occurs at specific reorder points determined by get_reorder_point(), which generates binary indicators for reorder days based on a fixed interval (i.e. every 10 days).
- Future demand and lead times are drawn from random distributions using get_demand() and get_lead_time().

Safety Stock Calculation

get_safety_stock() computes a buffer based on historical demand and lead times to ensure sufficient inventory during uncertainties.

Optimal Replenishment Quantity

For each reorder point, the script calculates the optimal reorder quantity using the following formula:

$$\begin{aligned}\text{Optimal Quantity} = \max(&\text{Cumulative Demand} + \\ &\text{Demand from Current Day to Next Reorder Point} - \\ &\text{Current Inventory}, 0)\end{aligned}$$

Dynamic Features for Training

Features such as inventory, demand, lead_time, reorder_point, and day are extracted and structured into sequences of length seq_len (5 days). These sequences represent the inputs to the model. The target variable, reorder_qty, is also prepared in sequence form.

2. Model Creation and Training

DCNN Model

The create_dcnn_model() function defines a Dilated Convolutional Neural Network (DCNN) with three dilated layers. This architecture is well-suited for sequence prediction tasks as it efficiently captures temporal patterns in time-series data.

Compilation and Training

- The model uses the Adam optimizer with a learning rate of 0.015 and the mean squared error (MSE) loss function.
- The data is divided into batches of size 64 and trained over 50 epochs.
- Checkpoints and TensorBoard callbacks are added to monitor and log the training process.

3. Model Fine-Tuning

After the initial training, the model undergoes additional fine-tuning:

- The model is reloaded using load_model().
- It is trained for 10 additional rounds, each with a single epoch, to further optimize performance on the training data.

4. Testing and Predictions

- Test data is prepared using the same feature extraction logic as the training data.
- Predictions are made using the trained model on the test data.
- Predictions (pred) are rounded to represent discrete reorder quantities.

Initialization

```
serialization_lib.enable_unsafe_deserialization()
DROPOUT_RATE = 0.01
MAX_SKU_ID = 100
MAX_STORE_ID = 50
BATCH_SIZE = 64
LEARNING_RATE = 0.015
EPOCHS = 50
seq_len = 5
b = 0.4
h = 0.4
model_file_name = "e2e.keras"
log_dir = os.path.join("logs", "scalars")
train_input_dynamic_ = []
train_output_ = []
cate_feature_ = []
```

- Enables unsafe deserialization, though this feature is risky for security.
- Defines several constants, including dropout rate, maximum SKU/Store IDs, batch size, learning rate, and number of epochs.
- Sets hyperparameters (b, h) for inventory management logic.
- Creates lists (train_input_dynamic_, train_output_, cate_feature_) to store processed training data.

Data Generation

```
for sku in range(1, MAX_SKU_ID + 1):
```

- Iterates over SKU_ID, simulating inventory data for each SKU

Simulating Inventory Data for each DC

```
for store in range(1, MAX_STORE_ID + 1):
```

- Nested loop iterating over store IDs. For each store:
  - Simulates initial inventory, lead times, and demand.
  - Sets up the simulation horizon (60 days) and interval for reordering (10 days).

Calculating Inventory Metrics

```
reorder_point_arr = get_reorder_point(horizon, interval)
n_reorder_pts = len(np.where(reorder_point_arr)[0])
history_lead_time = get_lead_time(mu_lead_time, std_lead_time, 50)
history_demand = abs(get_demand(mu_demand, std_demand, 50))
future_lead_time = abs(get_lead_time(mu_lead_time, std_lead_time, n_reorder_pts))
future_demand = abs(get_demand(mu_demand, std_demand, horizon))
safety_stock = get_safety_stock(history_lead_time, history_demand, service_level)
```

- get_reorder_point: Determines reorder points within the simulation horizon.
- get_lead_time & get_demand: Generate synthetic lead time and demand data based on normal distributions (mu, std).
- get_safety_stock: Computes safety stock based on historical lead time and demand.

Simulating Inventory Levels

```
inventory_level = np.zeros(horizon)
inventory_level[0] = initial_inventory
po_reaching = list(np.where(reorder_point_arr)[0] + np.int32(np.ceil(future_lead_time)))

if po_reaching[-1] < horizon:
    po_reaching.append(po_reaching[-1] + int(np.ceil(np.diff(po_reaching).mean())))
    optimal_rq = np.zeros(horizon)
    for i in range(1, horizon):
        ...
        optimal_qty = max(cum_demand + np.sum(future_demand[i:vm]) - inventory_level[i], 0)
        inventory_level[vm] += optimal_qty
        optimal_rq[i] = optimal_qty
        t += 1
```

- Initializes inventory levels and tracks future inventory via po_reaching (Purchase Order Reaching Days).
- Updates inventory levels based on demand and calculates optimal reorder quantities.

Creating Training Data

```
dynamic_features = ['inventory', 'demand', 'lead_time', 'reorder_point', 'day']
n = horizon - seq_len + 1
train_input_dynamic = np.empty((n, seq_len, n_dynamic_features))
train_output = np.empty((n, seq_len))
```

- Transofrms simulated inventory data into tabular form
- Extracts dynamic features (inventory, demand, lead time) and target (optimal reorder quantity)
- Converts data into sequences of length seq_len for training
- Aggregates dynamic input, output, and categorical features into global lists for batching

Combining Training Data

```
train_output_final = np.concatenate(train_output_)
train_input_dynamic_final = np.concatenate(train_input_dynamic_)
cate_feature_final = np.concatenate(cate_feature_)
```

- Merges training data across all SKUs and DC

Building and Training the Model

```
model = create_dcnn_model(...)
adam = optimizers.Adam(learning_rate=LEARNING_RATE)
model.compile(loss="mse", optimizer=adam, metrics=["mse", "mae"])
...
history = model.fit(...)
```

- Defines a DCNN model using create_dcnn_model
- Complies it with Adam optimizer and MSE loss
- Trains the model on the generated data

Iterative Training

```
for r in range(10):
    model = load_model(model_file_name)
    history = model.fit(...)
```

- Finetune the model for additional epochs, and saving the best checkpoint

Inference

```
test = train_input_dynamic
pred = np.round(model.predict([test, cate_feature_final[0:56]]))
pred[:, 0][np.where(reorder_point_arr)[0]]
```

- Prepares test data and makes predictions