2AMS40: Optimal Decision Making & Reinforcement Learning

Assignment 1

Mourad Boustani (1463233) & Danila Solodennikov (1954784)

1 Task 1

The optimal order size is 28. We calculated this by computing a weighted average of expected net profit for every order size, and picking the order size with the highest weighted average. The weighted average of an order size i is calculated by going over every possible demand, calculating its expected net profit when using order size i, and then weighing it with the probability of that demand occurring. A binomial distribution of demand (d) with a mean (μ) of 25 items and a variance (σ^2) of 20 items, points to a probability of "winning" (p) of 0.2 and 125 trials (n). This is calculated as follows. The formulas used are

$$p = \frac{\mu}{n}$$

and

$$n = \frac{\mu^2}{\mu - \sigma^2}$$

Let P_i be the expected net profit of order size i, with order sizes ranging from 0 to 125. Then:

$$P_i = \sum_{j=0}^{125} P[demand = j] \cdot ((min(i, j) \cdot 2000) + (i \cdot 400) + (max(0, i - j) \cdot 150))$$

- with j, the current demand and $P[demand = j]$, following from the binomial distribution $B(125, 0.2)$.
- with $min(i, j) \cdot 2000$ representing the revenue. If i is the order size and j is the demand, then we know we can either sell i items when the order size is smaller than the demand, or j items when the demand is smaller than the order size.
- with $i \cdot 400$ representing the cost of purchasing the order.
- with $max(0, i - j) * 150$ representing the cost of returning. if i is the order size and j is the demand, the items that are left will be $i - j$. Since we cannot return a negative amount of items, we set the value to 0 when $j > i$.

$\arg\max_i P_i = 28$, thus the optimal order size is 28

2 Task 2

Full code is provided in the appendix. To generate this synthetic data, we model the situation described in the assignment. We will make a simulation of 30 days. We iterate over the 30 days, starting with an initial inventory of 40 and a daily order size of 4. To simulate a day, we first get the order that is to be received on this day and add it to the current inventory. If the inventory is > 40, we set the inventory to the capacity of the storage (40), as we cannot hold more inventory than the capacity allows. We then sample a demand from the given binomial distribution. If the demand is greater than the current inventory, we set the amount sales to the current inventory, as we cannot sell more items than we have in inventory. We then subtract the amount of sales from the inventory. Then we calculate the revenue and the costs. The revenue is calculated by multiplying the amount of sales with the selling price of \$2000. The cost is the daily order (4) multiplied with the purchase cost of \$400. The holding cost is the inventory (that's left after all the sales) multiplied with the holding cost of \$150. Then, we put a new order in transit (of size 4) and append the collected data to the dataframe. The following is one of the simulations of the inventory (at the end of the day) we ran and an accompanying plot: $[17, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$.
As you can see, the inventory quickly stalls after the first day, due to the daily order size being 4 and the demand being 25 on average.

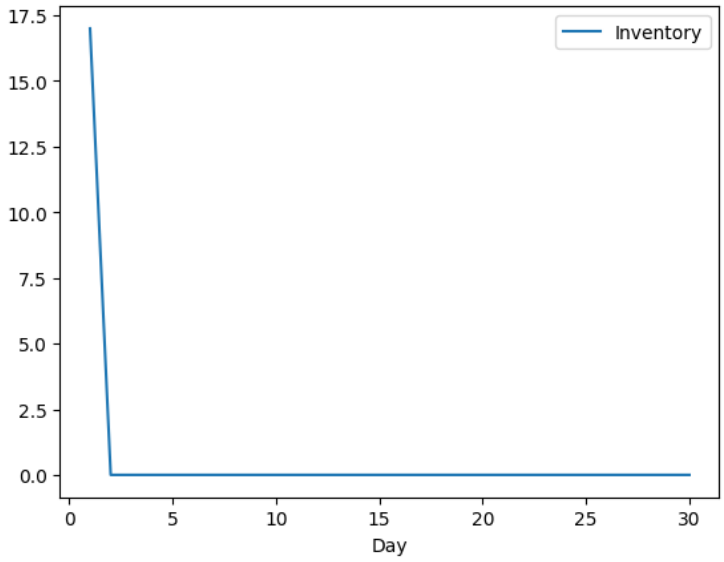

Fig. 1. A plot of a simulation of the inventory at the end of the day

3 Task 3

We formulate the MDP $= (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$

- $\mathcal{S}$, the finite set of states
- $\mathcal{A}$, the finite set of actions
- $\mathcal{P}$, the state transition probability matrix
- $\mathcal{R}$, the reward function
- γ, the discount factor

We define a state in the state space as a tuple (a, b). a being the inventory at the start of the day after receiving the order that is to be arrived at this day and b being the order placed yesterday and arriving tomorrow. This way we can properly define the transitions between the states.

$$\mathcal{S} = \{(a, b) \mid a \in \{0, 1, \ldots, 40\};\ b \in \{0, 1, \ldots, 10\}\}$$

We define the action space as the possibilities of orders that can be placed, therefore:

$$\mathcal{A} = \{0, 1, \ldots, 10\}$$

To define the probability matrix, we formulate the probability between any two states and an action:

$$\mathcal{P}^a_{ss'} = \mathbf{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$$

If $s' = (a', b')$ and $s = (a, b)$. Then the probability of this transition is the probability of the demand $d = -a + b - a'$, which is given by the binomial distribution $B(120, 0.2)$. This equality arises from the fact that a' reflects the result of whatever happened in state s. a' The action chosen does not affect the probability, and is therefore the same across actions, thus

$$\mathcal{P}_{(a,b);(a',b')} = \mathbf{P}[d = a + b - a']$$

$$\mathcal{R}^s_a = \mathbf{E}[R_{t+1} \mid S_t = s,\ A_t = a]$$

Let R_{t+1} be the net profit or loss made between the start of the current day and the start of the next day, as a result of the current action and state. This includes the revenue from sales, cost of ordering and holding costs of the current day. Let $s = (a', b')$, to avoid confusion with the action a.

$$\begin{aligned}\mathcal{R}^a_s = \mathbf{E}[&\text{Revenue from sales} - \text{Cost of ordering today}\\ &- \text{Holding costs} \mid S_t = (a', b'), A_t = a]\end{aligned}$$

$$\mathcal{R}^a_s = \mathbf{E}[(2000 \cdot min(demand, a')) - (400 \cdot a) - 150 \cdot (a' - min(demand, a')]$$

$$\begin{aligned}\mathcal{R}^a_s = \sum_{d=0}^{125} \mathbf{P}[demand = d] \cdot ((2000 \cdot min(demand, a'))\\ - (400 \cdot a) - (150 \cdot (a' - min(demand, a'))))\end{aligned}$$

$$\gamma = 0.9$$

Bellman equations

Value function $V(s)$

$$V^*(s) = \max_{a \in \mathcal{A}} \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a V^*(s') \right]$$

Action-value function $Q(s, a)$

$$Q^*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a' \in \mathcal{A}} Q^*(s', a')$$

4 Task 4

To compute the total expected discounted cost, we run a simulation of a 100 days, 10000 times. In a simulation, we start with the full inventory. For a hundred days, we receive the order that is to be received today, we sample a demand from the distribution and compute the revenue and incurred costs. We discount the cost with the current factor and sum it to the total discounted cost. We then place a new order in transit and discount the factor for the next day.

After all the simulations, we compute the mean and the standard deviation of the total costs of all simulations. Using that, we also compute a 99% confidence interval of the mean:

Total Expected Discounted Cost: -123520$ (99% C.I.: -123562, -123478)

Which proves this ordering policy to be profitable, however, intuitively, not optimal, as you are missing a lot of sales due to missed demand.

To gain such accuracy, we simulated episodes with rewards and after that computed standard deviation with confidence interval

5 Task 5

The optimal policy for every state is 10. Using small number of runs = 5 and delta = 1e-400, we achieved that the best policy will always be 10 and values approximately 107.556.094. This is why, 5 iterations can be already enough to get baseline results.

6 Task 6

Having max policy iteration = 1000, max value iteration = 1000 with number of runs = 5 and delta = 1e-400 we achieved this result: ...

While Policy iteration (PI) starts with a random policy, Value iteration (VI) starts with a value function. Even though that value VI algorithm is simple, it

is way slower that PI. While PI can make few iterations to converge, VI takes more iterations to converge too.

7 Appendix

7.1 Workload

The Task 1 and the task 2 was done by Mourad. Tasks 3,4,5,6 were done together with Danila and Mourad

7.2 Code

```
#Task 1
import scipy
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

purchase_cost = 400
price = 2000
return_cost_price = 150
mean = 25
variance = 20

n = (mean**2)/(mean - variance)
p = mean/n

possible_order_sizes = range(int(n) + 1)

def expected_net_profit(order_size, n, p,
    purchase_cost, price, return_cost_price):
    profit = 0
    for demand in possible_order_sizes:
        sold_items = min(order_size, demand)
        unsold_items = max(0, order_size - demand)
        probability = scipy.stats.binom.pmf(demand, n, p)
        revenue = sold_items * price
        cost = order_size * purchase_cost
        return_cost = unsold_items * return_cost_price
        sol = probability * (revenue - cost - return_cost)
        profit += sol
    return profit

profits = [expected_net_profit(order_size, n, p, purchase_cost,
           price, return_cost_price) for order_size in possible_order_sizes]
optimal_order_size = possible_order_sizes[np.argmax(profits)]

print(f"Optimal order size: {optimal_order_size}")
print(f"Expected profit: {max(profits)}")
```

```
plt.plot(possible_order_sizes, profits)

# Task 2
gamma = 0.9  # Discounting factor
initial_inventory = 40  # Starting inventory
daily_order = 4  # Daily order quantity
max_order_size = 10  # Maximum order size
storage_capacity = 40  # Storage capacity
lead_time = 2  # Lead time in days
purchase_cost = 400  # Cost per item
selling_price = 2000  # Selling price per item
holding_cost = 150  # Holding cost per unsold item
num_days = 30  # Number of days to simulate

# Initialize variables
inventory = initial_inventory
orders_in_transit = [0] * lead_time  # Orders in transit for each day

# Data collection
data = []

# Sample demand from given binomial distribution
def binomial_demand():
    return np.random.binomial(n, p)

# Re-run the simulation loop
data = []
inventory = initial_inventory
orders_in_transit = [0] * lead_time  # Reset orders in transit

for day in range(1, num_days + 1):
    # Receive order from two days ago
    inventory += orders_in_transit.pop(0)
    # Limit inventory to storage capacity
    inventory = min(inventory, storage_capacity)

    # Generate demand and calculate sales
    demand = binomial_demand()
    sales = min(demand, inventory)
    inventory -= sales

    # Calculate revenue and costs
    revenue = sales * selling_price
```

```
        cost = daily_order * purchase_cost
        holding_cost_total = inventory * holding_cost

        # Update orders in transit
        orders_in_transit.append(daily_order)

        # Record data
        data.append({
            'Day': day,
            'Inventory': inventory,
            'Demand': demand,
            'Sales': sales,
            'Revenue': revenue,
            'Cost': cost,
            'Holding Cost': holding_cost_total,
            'Net Profit': revenue - cost - holding_cost_total
        })

    # Convert data to DataFrame
    df = pd.DataFrame(data)

    plt.plot(df['Day'], df['Inventory'])
    df.head()


    #Making reward matrix

    import numpy as np
    import math
    import statistics,scipy
    #from statistics import sum

    # Define constants
    c = 400  # Cost of purchasing an item
    p = 2000  # Price of selling an item
    r = 150  # Holding cost per unsold item

    def get_reward_matrix():
        array = np.zeros([41,11])
        for i in range(len(array)):
            for j in range(len(array[i])):
                    temp_array=[]
                    for k in range(125):
                        # Calculate the future inventory after placing the replenishı
                        future_inventory = max(i -k,0)+j
```

```
                    # Calculate the number of sold items
                    sold_items = min(i, k)

                    # Calculate the purchase cost
                    purchase_cost = c * (sold_items)

                    # Calculate the revenue from sales
                    revenue = p * sold_items

                    # Calculate the holding cost for unsold items
                    holding_cost = r * (future_inventory - sold_items)

                    # Calculate the total reward
                    total_reward = purchase_cost + revenue - holding_cost

                    # Update the reward matrix


                    temp_array.append(scipy.stats.binom.pmf(k,125,0.2) * total_re


                array[i][j]=sum(temp_array)
    return array


#Making reward matrix Mourad

import numpy as np
import math
import statistics,scipy
#from statistics import sum

# Define constants
c = 400  # Cost of purchasing an item
p = 2000  # Price of selling an item
r = 150  # Holding cost per unsold item

def get_reward_matrix():
    array = np.zeros([41*11,11])
    for i in range(41):
        for j in range(11):
            for k in range(11):
                reward_per_demand = []
                for l in range(125):
                    prob_demand = scipy.stats.binom.pmf(l,125,0.2)
                    sales = min(i, l)
```

```
                    ordered_items = k
                    unsold_items = max(0, i - l)
                    revenue = sales * p
                    cost = ordered_items * c
                    holding_cost_total = unsold_items * r
                    total_reward = prob_demand * (revenue - cost - holding_cost_
                    reward_per_demand.append(total_reward)

                array[i*11+j][k]=sum(reward_per_demand)
    return array

#Task 4
import numpy as np
from scipy.stats import binom, norm


# Parameters
n = 125  # Number of trials in the binomial distribution for demand
p = 0.2  # Probability of success (demand) in each trial
selling_price = 2000  # Price per carton
cost_per_order = 400  # Cost per carton ordered
holding_cost = 150  # Holding cost per carton
order_quantity = 4  # Number of items ordered per day
initial_inventory = 40  # Initial inventory
gamma = 0.9  # Discount factor
lead_time = 2  # Lead time in days


# Function to simulate one episode
def simulate_episode():
    inventory = initial_inventory
    total_discounted_cost = 0
    discount_factor = 1
    orders_in_transit = [0] * lead_time  # Orders in transit for each day

    for _ in range(100):  # Simulating for 100 days
        # Generate demand
        demand = binom.rvs(n, p)

        # Calculate sales and update stock
        # Receive order from two days ago
        inventory += orders_in_transit.pop(0)
        inventory = min(inventory, 40)  # Limit stock based on storage capacity
        sales = min(inventory, demand)
        inventory -= sales
```

```
        # Calculate costs
        revenue = sales * selling_price
        order_cost = order_quantity * cost_per_order
        holding_cost_total = max(0, inventory) * holding_cost

        # Update total cost with discounting
        daily_cost = order_cost + holding_cost_total - revenue
        total_discounted_cost += discount_factor * daily_cost

        # Place an order and discount factor for next day
        orders_in_transit.append(order_quantity)
        discount_factor *= gamma

    return total_discounted_cost

# Run multiple episodes to compute mean and standard deviation
num_episodes = 10000  # Number of episodes
costs = [simulate_episode() for _ in range(num_episodes)]
mean_cost = np.mean(costs)
std_dev_cost = np.std(costs)

# Compute 99% confidence interval
z_score = norm.ppf(0.995)  # 99% confidence level
margin_error = z_score * std_dev_cost / np.sqrt(num_episodes)
confidence_interval = (mean_cost - margin_error, mean_cost + margin_error)

print("Mean Total Expected Discounted Cost:", mean_cost)
print("99% Confidence Interval:", confidence_interval)
```

```
#Task 5
#Value iteration

import numpy as np
from .reward_matrix_mourad import get_reward_matrix
from scipy.stats import binom, norm


# Initialize Markov Decision Process model
maxCapacity = 40
# State space: inventory levels and incoming stock
states = [(a, b) for a in range(41) for b in range(11)]
actions = list(range(11))  # Action space: number of items to order
demands = range(0,41) #demand space
rewards = get_reward_matrix()  # Direct rewards per state
gamma = 0.9  # discount factor

max_iter = 5
delta = 1e-400
V = np.zeros([41*11,11])
pi = np.zeros([41*11])

def transition_probability(s, action, s_prime):
    a, b = s
    a_prime, b_prime = s_prime

    demand = a + b - a_prime

    if demand < 0:
        return 0

    return binom.pmf(demand, 125, 0.2)

# Start value iteration
for i in range(max_iter):
      # Initialize max difference
    V_new = np.zeros([41*11,11]) # Initialize values
    for s in states:
        max_diff = 0
        for act in actions:
            val = rewards[s[0] * 11 + s[1]][act]  # Get direct reward
            for a in actions:
                max_val = 0
                for demand in demands:
                    new_inventory = min(s[0] - demand + s[1], maxCapacity)
```

```
                    if new_inventory < 0:
                        break
                    s_prime =  (s[0] + s[1] - demand, a)
                    prob = transition_probability(s, a, s_prime)
                    val += prob * (gamma * V[s_prime[0]][s_prime[1]])

                # Store value best action so far
                max_val = max(max_val, val)

            # Update best policy
            if V[s[0] * 11 + s[1]][act] < val:
                # Store action with highest value
                pi[s[0] * 11 + s[1]] = actions[a]

            # Update value with highest value
            V_new[s[0] * 11 + s[1]][act] = max_val
            max_diff = max(max_diff,
                           abs(V[s[0] * 11 + s[1]][act] -
                           V_new[s[0] * 11 + s[1]][act]))

    # Update value functions
    V = V_new

    # If diff smaller than threshold delta for all states, algorithm terminates
    if max_diff < delta:
        break

#Task 6
import numpy as np
from .making_reward_function import get_reward_matrix

# Supply chain homework - policy iteration


# Set policy iteration parameters
max_policy_iter = 100  # Maximum number of policy iterations
max_value_iter = 100  # Maximum number of value iterations

# Initialize Markov Decision Process model
maxCapacity = 40
actions = range(0,10+1) #action space
states = range(0,maxCapacity+1) #state space
demands = range(0,41) #demand space
rewards = get_reward_matrix()  # Direct rewards per state
gamma = 0.9  # discount factor
```

```
max_iter = 5
delta = 1e-400
V = np.zeros([41,11])
pi = np.zeros([41,11])



for i in range(max_policy_iter):
    # Initial assumption: policy is stable
    optimal_policy_found = True

    # Policy evaluation
    # Compute value for each state under current policy
    for j in range(max_value_iter):
        max_diff = 0  # Initialize max difference

        for s in states:
            for act in actions:

                for a in actions:
                    max_val = 0
                    # Compute state value
                    val = rewards[s][act]  # Get direct reward
                    for demand in demands:
                        metDemand = min(s, demand); #compute met demand
                        unmetDemand = max((demand-s), 0); #compute unmet demand

                        s_next = min(s-metDemand+a, maxCapacity) #update state

                        val += (1/5) * (gamma * V[s_next][act])
# Add discounted downstream values

                    max_diff = max(max_diff, abs(val - V[s][act]))
                    V[s][act] = val
            if max_diff < delta:
                break

    # Policy iteration
    # With updated state values, improve policy if needed
    for s in states:
        for act in actions:
            val_max = V[s][act]
            for a in actions:
                val = rewards[s][act]  # Get direct reward
                for demand in demands:
```

```
                    metDemand = min(s, demand); #compute met demand
                    unmetDemand = max((demand-s), 0); #compute unmet demand

                    s_next = min(s-metDemand+a, maxCapacity) #update state

                    val += (1/5) * (gamma * V[s_next][act])
# Add discounted downstream values

                # Update policy if (i) action improves value and
                # (ii) action different from current policy
                if val > val_max and pi[s][act] != a:
                    pi[s][act] = a
                    val_max = val
                    optimal_policy_found = False

        # If policy did not change, algorithm terminates
        if optimal_policy_found:
            break
```