**Catholic University Eichstätt-Ingolstadt Faculty of Business and Economics**

**Assistant Professorship of Operations Management**

**Prof. Dr. Pirmin Fontaine, Daniel Müllerklein**

**Reinforcement Learning - Supply Chain Ordering Management: An application to the beer game**

**– Project work –**

submitted by

Dennis Götz

Vincent Bläske

Paul Posselt

Serkan Akdemir

# Table of contents

[Table of contents I](#_bookmark0)

[List of illustrations II](#_bookmark1)

[List of tables III](#_bookmark2)

[List of abbreviations IV](#_bookmark3)

1. [Introduction and structure of the paper 1](#_bookmark4)
2. [Problem description 1](#_bookmark5)
3. [Data generation 3](#_bookmark10)
4. [Implementation 7](#_bookmark21)
   1. [Implementation of Q-learning algorithm 7](#_bookmark22)
   2. [Moving to DQN 8](#_bookmark23)
5. [Discussion and results 9](#_bookmark24)

[Bibliography V](#_bookmark29)

[Appendix VI](#_bookmark30)

List of figures

[Figure 1: Four test problem data 4](#_bookmark14)

[Figure 2: Determinanten des Kundenwerts 4](#_bookmark15)

[Figure 3: Abbildungsbeispiel einer eigens erstellten Abbildung 5](#_bookmark16)

# Introduction and structure of the paper

In order to illustrate the ideas of dynamic systems, Jay Forrester of MIT first created the iconic Beer Distribution Game in the late 1950s. In this instance, the dynamic system is a supply chain that transports beer from a brewery to the final customer. The supply chain structure and game rules are straightforward, but the ensuing behavior is highly complex, which is what makes the game so exciting.

A way to creating simulation models of real-world systems is agent-based modeling. Agents that are used in agent-based modeling capture entities and exhibit behavior in response to internal or external events. In the beer distribution game, the players will be represented by agents. Instead of teaching the agents a set of predefined behaviors, the agents will be trained to play the beer distribution game using machine learning techniques, particularly a technique known as Reinforcement Learning. Reinforcement learning emphasizes interaction-based learning. The agent must experiment with different actions to determine which ones have the best results rather than being instructed which to do. One approach to reinforcement learning is Q-learning.

In this paper, the goal is to implement a Q-learning algorithm to deal with inventory management challenges such as changing demands, varying lead times, and/ or disruptions.

The structure of the essay is as follows: The reinforcement learning problem is briefly described in section 2. The generated data to test and train the algorithm is shown in section 3. In section 4, the Q-learning algorithm will be implemented and adjusted to account for Deep Q-networks. In section 5, the findings on model performance after testing different parameters will be summarized and it will be discussed, on how well reinforcement learning suits the applied problem setting.

# Problem description

The problem at hand is a minimization problem. The goal is to keep costs to a minimum within a defined supply chain. Within the supply chain, each actor (producer, supplier, etc.) has an agent that can trigger an order at the upstream actor. Each of these agents tries to minimize not only the costs incurred by itself, but the total sum of all costs incurred by the actors in the supply chain. Costs can arise both from the stockpiling of products and from unsatisfied demand. In every level of the supply chain at each time step, four events happen:

1. Previous orders are received (according to the lead-times) from the upstream actor.
2. Order size received from the downstream actor.
3. The received order is fulfilled from on-hand inventory (if possible).
4. The Agent decides about placing order for stock replenishment.

The supply chain represents the system, which can assume different states. These states result

from the inventory position of the individual actors at a specified time and each state is represented by a vector with these values as elements. Since the individual elements of the vector can theoretically assume arbitrarily large or small values, the values which represent the inventory position need to be transformed into a coding first to generate a limited number of possible states. This results in n over k (binomial coefficient) different possible states, where n represents the number of different coded states and k represents the number of different actors within the supply chain. The initial state for the system is {12, 12, 12, 12}, meaning that each actor has an inventory position of 12 units.

In this problem, the agent has the possibility to react to the environmental state of the supply chain by placing an order with the upstream actor. The ordered quantity is represented by two different variables. The variable X represents the demand of the downstream actor. If, for example, the customer places an order with the retailer for 10 units, the variable X for the retailer takes the value 10. In turn, the variable Y indicates the deviation of the own order at the upstream actor from the order received from the downstream actor. This means that if the value of Y is -1, the agent will order one less unit from the upstream actor than was requested from it by the downstream actor. With a value of +1, the agent orders one unit more, and with a value of 0, it orders exactly as many units as are ordered from it by its downstream actor. The agents can now set the values for Y as an action. These values for Y can theoretically be between minus X and plus infinity. Simulations can be used to find out to which range the values for Y can be fixed without influencing the result in any essential way.

The transition probabilities are based on the Q-values, whereby, since this is a minimization problem, the action that promises the lowest Q-value is always chosen. At the beginning, the probabilities for each action are the same. By estimating the Q-values, new values are gradually learned for the different actions, resulting in different transition probabilities. In this learning phase, the agent selects with a certain probability the action that promises the lowest Q-value (exploitation) and with a certain probability a randomly chosen action (exploration). The initial probabilities are 0.98 for exploration and 0.02 for exploitation. They always add up to 1. The agent therefore is more likely to explore first because his lack of knowledge about the environment. The relationship between exploitation and exploration shifts during the learning process, with the probability of exploitation increasing linearly and the probability of exploration decreasing linearly. After completing the learning process, the agent applies a greedy search and according to the systems state always chooses the action that grants him the lowest Q value.

The reward the agent receives from the environment is calculated using a reward function. In this function, both the inventory position and the backorder are taken into account. These two values are weighted with a cost factor each, alpha and beta, and summed up. It is important here that the costs for all actors involved in the supply chain are calculated and then completely aggregated. This results in an agent wanting to minimize not only the costs incurred within their own operations but also the costs of all parties involved in the supply chain. The two cost factors alpha and beta can be individually chosen here in such a way that they correspond to the costs incurred in the real system.

# Data generation

For the first main test problem, we use the customer demands and lead time data from Kimbrough et al. For test problems 1-3, we use the data generated by Chaharsooghi et al. The parameters were introduced by Kimbrough et al: Customer demand uniformly distributed between [0,15] and lead-times uniformly distributed from 0 to 4.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Figure 1: Four test problem data: include main test problem and three new generated test problem: Kimbrough et al. 2002, P. 328-329

For our own test problem we define a list of random variables for the customer demands and the lead times. The random values for the customer demands can take the integers 0 to 15. For the lead times, integers in the range 0 to 4 are generated. The size of the lists is 35 for both and represents the number of calendar weeks.

# Implementation

Lorem ipsum

## Implementation of Q-learning algorithm

Lorem ipsum

## Moving to DQN

As described in the paper, using an infinite space for the elements in the state vector, it is impossible to determine the near-optimal policy, when using the Q learning approach. This is because it would need infinite search power, as the Q table would grow infinitely. Thus, the paper proposes to discretize the state space to some coded values. However, in the context of the beer game, relying discretized states can be restrictive, as this would mean information loss. It would in consequence be better to represent the state variable, ranging from negative infinity to positive infinity.

Using DQN permits us to handle these scenarios where the number of states aligns more realistically with real-world situations. In our specific case this implies the adoption of a continuous state space for the actors. Instead of the building of a Q table, we would need to find a function to approximate the Q values. This approximation can be achieved through the use of a neural network, that predicts a Q value based on a current state.

To understand better what needs to be changed on the current implementation of the inventory optimization, we will explain how the DQN works.

To begin with, a replay buffer interacts with the environment. In these interactions, it collects data by following an ε-greedy policy for specific state-action pairs. This data collection process is focused on obtaining both the reward and the state that follows a particular action. Once acquired, this data is used as training data, for later use. From training data amassed, a sample is made to create a subset. This subset, becomes the basis for training of a neural network, called the Q neural network. This network is used to predict the Q value of the current state. In parallel, a second neural network, the Target network, predicts the Q value of the next state, that is linked to the action undertaken after the current state. With the neural network architecture in place, the loss is computed in the next step. This metric measures the discrepancy between the predicted Q value from the Q neural network and the target Q value. The target Q value is computed by adding the reward associated with the current action and the next states Q value. This discrepancy could for example be calculated through the Mean Squared Error (MSE). This loss is then used for backpropagation, a process to adjust the neural networks weights with the goal to minimize the loss between the current Q value and the target Q value. Backpropagation works by plotting the gradients of the loss, with respect to the network weights, which in turn indicates how much the weights have to be adjusted to reduce the loss. The extent of the update is controlled by the learning rate α, ensuring a balanced convergence toward more accurate Q value predictions. However, the update of the weights is only carried out for the Q NN and not the target NN, to ensure stability in the learning process. By keeping the target NN's weights fixed for a set number of iterations, we provide a more stable and consistent target for the Q NN to learn from. Although the update of the weights of the target NN happen periodically, by matching the weights to the weights of the Q NN. This whole process is repeated for each time step until a specified maximum number of iterations is reached.

Following this logic, we can now understand what needs to be changed in our implementation to apply the DQN. As already mentioned, the state space doesn’t have to be encoded anymore and we can just assume the infinite state space. Thus, the state can be represented as a vector of continuous values [S1, S2, S3, S4]. This is also recommended as it wouldn’t make sense to use DQN if the state and action space stays the same, as the training of the NN takes more resources, when the state and action space is small.

For the action space, it has to remain discretized. This is because the neural network needs a defined number of output neurons, each corresponding to a specific action. Nevertheless, DQN allows for a broader action space without employing too many resources, where in contrast, the increase of actions would increase the number of operations in a Q table exponentially.

Most importantly, we wouldn’t need a Q table anymore. We could just replace it with the neural network that reliably predicts the Q values. To train that NN, we would have to implement a new algorithm that updates the weights for the NN iteratively, to get more realistic Q values.

We would also need to be implemented is that of replay buffer. In this, the state action combination, paired with their reward and next state is stored for each step. This could be stored as a list with multiple “experiences” stored in them. As we have multiple actors in the beer game, we need to consider that each experience stored should capture the combined actions and states of all actors. Thus, each experience should be represented by a vector of states, action, rewards and next states. Additionally, the replay buffer should also include information about if the next state is terminal or not, as if the next state is terminal, there is no future reward to consider, and this would signify the end of an episode (35 weeks) and a new one can start. From this list we should be able to sample a batch of those experiences.

On top of the hyperparameters that we have in Q learning, we will get some more parameters to tune, such as the size of the replay buffer, the batch size of the subset for training the NN, the update frequency for the target network and the hyperparameters of the neural networks, including the learning rate for the backpropagation.

Although some of the elements wouldn’t need to be changed. Exploration/exploitation stays the same. We still use an ε-greedy strategy with ε decreasing over time, leading to exploitation in contrast to exploration.

# Discussion and results

Lorem Ipsu

# Bibliography

S. Kamal Chaharsooghi, J. Heydari, S. Hessameddin Zegordi, A reinforcement learning model for supply chain ordering management: An application to the beer game (2008) 949–959.

S.O. Kimbrough, D.J. Wu, F. Zhong, Computers play the beer game: can artificial agents manage supply chains? Decision Support Systems 33 (2002) 323–333.

# Appendix

Appendix 1: inventory beer game implementation code VII

Appendix 2: Variante b) – Anleitung: Abbildung als Grafik importieren XIII

Appendix 3: Fragebogenbeispiel (Auszug) XIV

**Appendix 1:**

# Inventory beer game implementation code

**import** numpy **as** np

**import** pandas **as** pd

**import** matplotlib.pyplot **as** plt

**import** random

**import** sys

*# Supply Chain and its agents*

supply\_chain **=** {'level 0': 'Customer',

'level 1': 'Retailer',

'level 2': 'Distributor'}

agents **=** [supply\_chain[i] **for** i **in** list(supply\_chain**.**keys())[1:]]

*# All possible coded inventory positions of agents and the respective state pairs (5 states with 25 state pairs)*

states **=** np**.**arange(start**=**1, stop**=**6)

state\_pairs **=** [(i, j) **for** i **in** states **for** j **in** states]

*# y-value in ordering rule x+y with x equal to the demand from the downstream agent and the respective action pairs (4 actions with 16 action pairs)*

actions **=** np**.**arange(stop**=**4)

action\_pairs **=** [(i, j) **for** i **in** actions **for** j **in** actions]

*# Initial matrix with Q-values (minimization -> high values)*

Q\_values **=** np**.**full(shape**=**(len(state\_pairs), len(action\_pairs)), fill\_value**=**1000)

df\_Q\_values **=** pd**.**DataFrame(data**=**Q\_values, index**=**state\_pairs, columns**=**action\_pairs)

display(df\_Q\_values**.**head())

*# Define parameters*

iteration **=** 1

max\_iteration **=** 1

T **=** 15 *#smaller*

gamma **=** 1 *#smaller*

alpha **=** 0.17

sigma **=** 0.05

proba\_exploitation **=** 0.02

start\_exploitation **=** proba\_exploitation

truck\_load **=** 4

train\_load **=** 4

*# Set up Lists to store the results of each iteration/episode*

S **=** [list(np**.**repeat(0, len(agents))) **for** i **in** range(T**+**1)]

CS **=** [list(np**.**repeat(0, len(agents))) **for** i **in** range(T**+**1)]

D **=** [list(np**.**repeat(0, len(agents))) **for** i **in** range(T**+**1)]

O **=** [list(np**.**repeat(0, len(agents))) **for** i **in** range(T**+**1)]

x **=** [list(np**.**repeat(0, len(agents))) **for** i **in** range(T**+**1)]

y **=** [list(np**.**repeat(0, len(agents))) **for** i **in** range(T**+**1)]

r **=** [list(np**.**repeat(0, len(agents))) **for** i **in** range(T**+**1)]

R **=** [0 **for** i **in** range(T)]

G **=** [0 **for** i **in** range(T)]

q **=** [0 **for** i **in** range(T)]

Q **=** [1] **+** [0 **for** i **in** range(max\_iteration)]

*# Define parameters*

iteration **=** 1

max\_iteration **=** 1

T **=** 15 *#smaller*

gamma **=** 1 *#smaller*

alpha **=** 0.17

sigma **=** 0.05

proba\_exploitation **=** 0.02

start\_exploitation **=** proba\_exploitation

truck\_load **=** 4

train\_load **=** 4

*# Set up Lists to store the results of each iteration/episode*

S **=** [list(np**.**repeat(0, len(agents))) **for** i **in** range(T**+**1)]

CS **=** [list(np**.**repeat(0, len(agents))) **for** i **in** range(T**+**1)]

D **=** [list(np**.**repeat(0, len(agents))) **for** i **in** range(T**+**1)]

O **=** [list(np**.**repeat(0, len(agents))) **for** i **in** range(T**+**1)]

x **=** [list(np**.**repeat(0, len(agents))) **for** i **in** range(T**+**1)]

y **=** [list(np**.**repeat(0, len(agents))) **for** i **in** range(T**+**1)]

r **=** [list(np**.**repeat(0, len(agents))) **for** i **in** range(T**+**1)]

R **=** [0 **for** i **in** range(T)]

G **=** [0 **for** i **in** range(T)]

q **=** [0 **for** i **in** range(T)]

Q **=** [1] **+** [0 **for** i **in** range(max\_iteration)]

*# helperfunctions*

*# function to convert real states to coded states*

**def** coded\_state(inventory):

**if** inventory **<** **-**5:

**return** 1

**elif** inventory **<** 0:

**return** 2

**elif** inventory **<** 5:

**return** 3

**elif** inventory **<** 12:

**return** 4

**else**:

**return** 5

*# function to view episodes*

**def** fun\_episode(S, CS, D, O, x, y, r, head**=True**):

time\_steps **=** ['t='**+**str(i) **for** i **in** np**.**arange(start**=**1, stop**=**T**+**2)]

df **=** pd**.**DataFrame({'Inventory/States S': [tuple(i) **for** i **in** S],

'Coded states CS': [tuple(i) **for** i **in** CS],

'Demand x': [tuple(i) **for** i **in** x],

'Distribution amount D': [tuple(i) **for** i **in** D],

'Action y': [tuple(i) **for** i **in** y],

'Ordering size': [tuple(i) **for** i **in** O],

'Costs r': [tuple(i) **for** i **in** r]},

index**=**time\_steps)

df**.**index**.**name **=** 'Time'

**if** head **==** **True**:

**return** display(df**.**head())

**else**: **return** display(df)

*# function to define the starting states of the supply chain*

**def** fun\_start\_state(how**=**'value', inv**=**12):

*# select own start values for all agents*

**if** how **==** 'value':

**for** agent **in** range(len(agents)):

S[0][agent] **=** inv

CS[0][agent] **=** coded\_state(S[0][agent])

*# each episode has a change of 50% to start with high (12) or with low (-2) inventory for all agents*

**elif** how **==** 'high/low':

starting\_state **=** random**.**choices(population**=**['high', 'low'], weights**=**[0.5, 0.5])[0]

**if** starting\_state **==** 'high':

**for** agent **in** range(len(agents)):

S[0][agent] **=** 12

CS[0][agent] **=** coded\_state(S[0][agent])

**else**:

**for** agent **in** range(len(agents)):

S[0][agent] **=** **-**2

CS[0][agent] **=** coded\_state(S[0][agent])

*# random starting positions (-10 to 16) of agents for each episode*

**elif** how **==** 'random':

**for** agent **in** range(len(agents)):

S[0][agent] **=** random**.**choices(np**.**arange(start**=-**10, stop**=**16))[0]

CS[0][agent] **=** coded\_state(S[0][agent])

*# function to measure the Q-value decrease*

**def** fun\_q\_decrease(alpha, q\_value, G):

new\_q\_value **=** (1**-**alpha) **\*** q\_value **+** alpha **\*** G

decrease **=** new\_q\_value **/** q\_value

**return** new\_q\_value, decrease

*# function to visualize the Q-vlaue decrease*

**def** plot\_q\_decrease(Q):

plt**.**figure(figsize**=**(12,4))

plt**.**plot(Q)

plt**.**xlim(0, max\_iteration)

plt**.**ylim(0, 1.1)

plt**.**xlabel('episode/iteration')

plt**.**ylabel('Q-value decrease')

plt**.**title('Q-value decrease per episode', size**=**18)

**return** plt**.**show()

*# Three possible options to set starting states*

fun\_start\_state(how**=**'value', inv**=**10)

print('Inventory state S: {}'**.**format(S))

print('Coded state S: {}\n'**.**format(CS))

fun\_start\_state(how**=**'high/low')

print('Inventory state S: {}'**.**format(S))

print('Coded state S: {}\n'**.**format(CS))

fun\_start\_state(how**=**'random')

print('Inventory state S: {}'**.**format(S))

print('Coded state S: {}\n'**.**format(CS))

Inventory state S: [[10, 10], [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, 0]]

Coded state S: [[4, 4], [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, 0]]

Inventory state S: [[12, 12], [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, 0]]

Coded state S: [[5, 5], [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, 0]]

Inventory state S: [[3, 1], [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, 0]]

Coded state S: [[3, 3], [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, 0]]

*# Main function*

**while** iteration **<=** max\_iteration:

*# if iteration in [1, 10, 100, 1000, 10000, 25000, 50000, 75000, 100000, max\_iteration]:*

*# print('iteration {}'.format(iteration))*

print('-------------------episode {}------------------'**.**format(iteration))

print('Exploitation probability: {}\n'**.**format(proba\_exploitation))

t**=**0

*# random starting position of agents for each episode*

fun\_start\_state(how**=**'value', inv**=**12) *#option 1: ('value', inv=12); option 2: 'high/low'; option 3: 'random'*

**while** t **<** T:

print('-----------------time step t={}----------------'**.**format(t))

*# view state and coded state*

state **=** tuple(S[t])

c\_state **=** tuple(CS[t])

print('State S: {} and coded state CS: {}'**.**format(state, c\_state))

*# define random lead time lag (between 0 and 2)*

lag **=** random**.**choices(population**=**[0, 1, 2], weights**=**[0.6, 0.3, 0.1])[0]

print('Lead time lag of {} time steps\n'**.**format(lag))

**for** agent **in** range(len(agents)):

level **=** supply\_chain['level ' **+** str(agent**+**1)]

print('------------------{}------------------'**.**format(level))

*# step 1: receive the new demand of the downstream agent*

**if** level **==** 'Retailer':

x[t][agent] **=** np**.**random**.**randint(low**=**0, high**=**15)

**else**:

x[t][agent] **=** O[t**-**1][agent**-**1]

*# Add negative inventory (=demand of previous time steps) to the new demand*

inventory **=** S[t][agent]

**if** inventory **>=** 0:

print('Demand x from downstream ({}) at t-1: {}'**.**format(supply\_chain['level ' **+** str(agent)], x[t][agent]))

**else**:

print('Demand x from downstream ({}) at t-1 + demand of previous time steps:'**.**format(supply\_chain['level ' **+** str(agent)]))

print( '{} + {} = {}'**.**format(x[t][agent], np**.**abs(inventory), x[t][agent] **+** np**.**abs(inventory)))

x[t][agent] **+=** np**.**abs(inventory)

*# step 2: fulfill order of downstream agent from onhand inventory and calculate possible backlog costs*

**if** inventory **>=** 0:

D[t][agent] **=** min(x[t][agent], inventory) *#distribution quantity*

print('Distribution size D: {}'**.**format(D[t][agent]))

**else**:

D[t][agent] **=** min(x[t][agent], D[min(0, t**-**1)][agent**+**1]) *#negative inventory is demand of previous time steps and still needs to be complied*

print('Distribution size D (delivery of upstream agent in t-1): {}'**.**format(D[t][agent]))

backlog **=** x[t][agent] **-** D[t][agent] *#penalty/backlog costs (previous backlogs included in demand if inventory is negative)*

print('Backlog size: {}'**.**format(backlog))

*# step 3: placing order for stock replenishment*

*# define best action for agent and select it with initially very small probability (first exploration and mostly random choices)*

best\_action **=** df\_Q\_values**.**iloc[df\_Q\_values**.**index**.**get\_loc(tuple(CS[t]))]**.**idxmin()[agent]

print('Best action y\*: {}'**.**format(best\_action))

y[t][agent] **=** random**.**choices([best\_action] **+** list(actions), weights**=**[proba\_exploitation] **+** list(np**.**repeat((1 **-** proba\_exploitation) **/** len(actions), len(actions))))[0]

print('Action y: {}'**.**format(y[t][agent]))

**if** inventory **>=** 0:

O[t][agent] **=** x[t][agent] **+** y[t][agent] *#O[t+lag] !!*

**else**: O[t][agent] **=** x[t][agent] **+** inventory **+** y[t][agent] *#subtract negative inventory (demand of previous time steps) again - has been ordered already*

print('Ordering size O: {}\n'**.**format(O[t][agent]))

*# step 4: previous orders are received from the upstream agent (update states for t+1)*

print('UPDATING STATES AND CALCULATING COSTS PER AGENT')

**for** agent **in** range(len(agents)):

level **=** supply\_chain['level ' **+** str(agent**+**1)]

print('------------------{}------------------'**.**format(level))

inventory **=** S[t][agent] **-** x[t][agent]

*#if level == 'Retailer': lag = 0 #retailer's lead time to customer is zero*

**if** level **!=** agents[**-**1]:

**if** t **==** 0:

S[t**+**1][agent] **=** inventory **+** truck\_load **+** D[t][agent**+**1] *#truck load represets order of t-2*

print('Inventory {} after receiving truck load +{} (demand of t-2) and order +{}: {}'**.**format(inventory, truck\_load, D[t][agent**+**1], S[t**+**1][agent]))

**elif** t **==** 1:

S[t**+**1][agent] **=** inventory **+** train\_load **+** D[t][agent**+**1] *#train load represets order of t-1*

print('Inventory {} after receiving train load +{} (demand of t-1) and order +{}: {}'**.**format(inventory, train\_load, D[t][agent**+**1], S[t**+**1][agent]))

**else**:

S[t**+**1][agent] **=** inventory **+** D[t][agent**+**1]

print('Inventory {} after receiving order +{} of t-1: {}'**.**format(inventory, D[t][agent**+**1], S[t**+**1][agent]))

**else**:

S[t**+**1][agent] **=** inventory **+** O[t][agent] *#last agent in supply chain has no supplier (delivers from warehouse)*

print('Inventory {} receiving order +{} of t-1: {}'**.**format(inventory, O[t][agent], S[t**+**1][agent]))

*# update coded states*

CS[t**+**1][agent] **=** coded\_state(S[t**+**1][agent])

*# calculate agent's costs (onhand inventory holding costs + penalty costs)*

r[t][agent] **=** 1 **\*** max(S[t**+**1][agent], 0) **+** 2 **\*** (x[t][agent] **-** D[t][agent]) *#backlog*

print('Costs r: {}\n'**.**format(r[t][agent]))

*# calculate the total supply chain costs in t*

action **=** tuple(y[t])

R[t] **=** np**.**sum(r[t])

print('Supply Chain costs R in state {} with action {} at t={}: {}'**.**format(state, action, t, R[t]))

*# increase time step t*

t **+=** 1

print('\n\n')

*# view last episode*

fun\_episode(S, CS, D, O, x, y, r, head**=False**)

*# Loop from T to t=0 to calculate immediate rewards and returns*

print('-----Rewards and Total discounted returns-----')

**for** t **in** np**.**arange(start**=**0, stop**=**T)[::**-**1]:

*# view immediate reward of each visited state-action-pair*

print('Immediate reward R in t={}: {}'**.**format(t, R[t]))

*# calculate the return G: total discounted rewards*

G[t] **=** R[t] **+** np**.**sum(R[t**+**1:] **\*** np**.**array([gamma**\*\***i **for** i **in** np**.**arange(start**=**1, stop**=**T**-**t)]))

*# view returns of all state-action-pairs*

print('\nTotal discounted rewards (Return G): \n{}\n'**.**format(G))

*# update all visited Q-values*

print('---------------UPDATING Q-VALUES--------------')

**for** t **in** range(T):

state **=** tuple(CS[t])

action **=** tuple(y[t])

*# get current Q-value*

q\_value **=** df\_Q\_values**.**iloc[df\_Q\_values**.**index**.**get\_loc(state), df\_Q\_values**.**columns**.**get\_loc(action)]

print('t={}: state: {}, action: {}, old Q-value: {}, return: {}'**.**format(t, state, action, q\_value, G[t]))

*# update Q-values according to equation 12 in the paper (slide 30 TD-learning script)*

df\_Q\_values**.**iloc[df\_Q\_values**.**index**.**get\_loc(state), df\_Q\_values**.**columns**.**get\_loc(action)] **=** (1**-**alpha) **\*** q\_value **+** alpha **\*** G[t] *#equal to: q\_value + alpha \* (G[t] - q\_value)*

new\_q\_value, decrease **=** fun\_q\_decrease(alpha, q\_value, G[t])

q[t] **=** decrease

print('New Q-value: {}\n'**.**format(new\_q\_value))

*# calculate mean Q-value decrease of current episode*

Q[iteration] **=** np**.**mean(q)

*# Check if the convergence criterion has been reached*

**if** (iteration **>** 100) **&** (np**.**mean(Q[iteration**-**5:iteration]) **<** sigma):

print('Convergence!')

**break**

*# increase exploitation probability linearly*

proba\_exploitation **+=** (0.95 **-** start\_exploitation) **/** max\_iteration

print('\n\n')

*# start with next iteration*

iteration **+=** 1

*# round dataframe with Q-values*

df\_Q\_values **=** np**.**round(df\_Q\_values, 2)

*# view new Q-value table*

display(df\_Q\_values)

*# plot decrease of Q-vlaues*

plot\_q\_decrease(Q)

# Honorary declaration

I hereby declare on my honor that I have prepared this thesis independently and without the use of any aids other than those indicated.

The work has not yet been submitted to any other examination authority and has not yet been published.

I am aware that an untrue declaration will have legal consequences.

……………….............. .........................

(Place) (Date)

........................................................................

(Signature)