ARTIFICIAL INTELLIGENCE A-Z

Learn How To Build An A.I

BY: Hadelin de Ponteves
and Kirill Eremenko

SUPER DATASCIENCE
MAKING THE COMPLEX SIMPLE

Table of contents

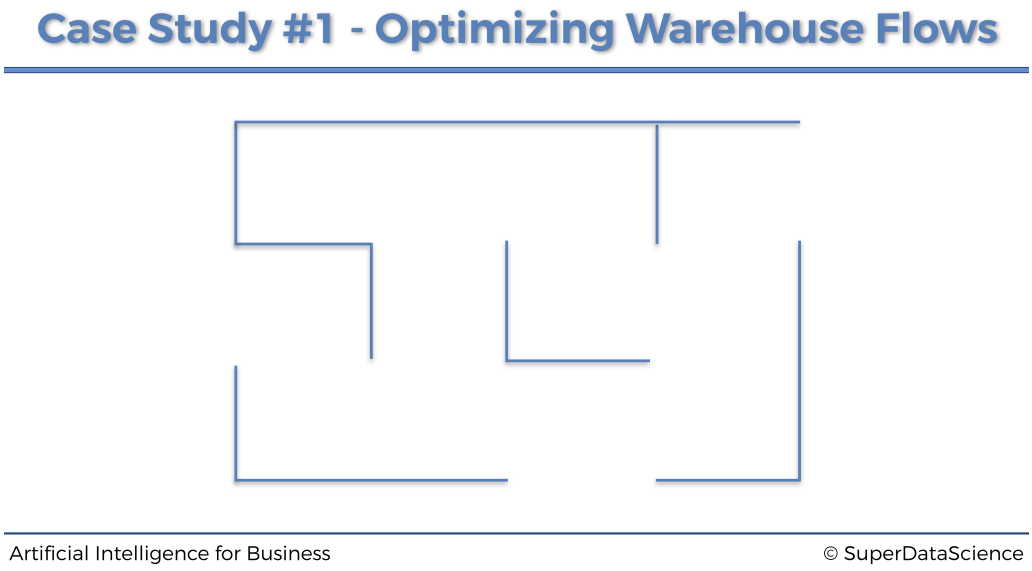
1 A Q-Learning Implementation for Process Optimization

Here we go with our first case study and our first AI model. We hope you are ready.

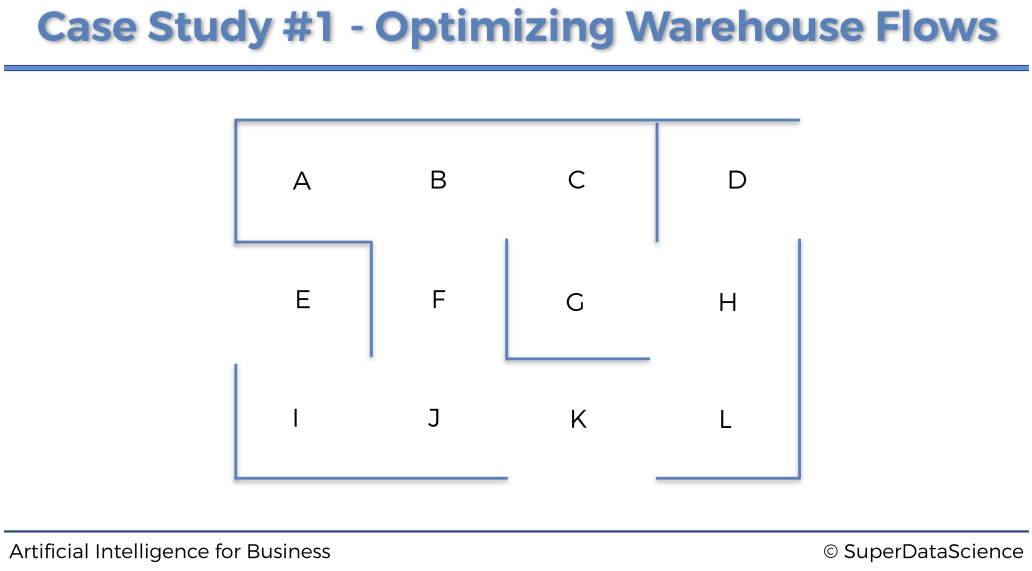
1.1 Case Study: Optimizing the Flows in an E-Commerce Warehouse

1.1.1 Problem to solve

The problem to solve will be to optimize the flows inside the following warehouse:



The warehouse belongs to an online retail company that sells products to a variety of customers. Inside this warehouse, the products are stored in 12 different locations, labeled by the following letters from A to L:



As the orders are placed by the customers online, an Autonomous Warehouse Robot is moving around the warehouse to collect the products for future deliveries. Here is what it looks like:

Figure 1: Autonomous Warehouse Robot

The 12 locations are all connected to a computer system, which is ranking in real time the priorities of product collection for these 12 locations. For example, at a specific time t, it will return the following ranking:

Priority Rank	**Location**
1	G
2	K
3	L
4	J
5	A
6	I
7	H
8	C
9	B
10	D
11	F
12	E

Location G has priority 1, which means it is the top priority, as it contains a product that must be collected and delivered immediately. Our Autonomous Warehouse Robot must move to location G by the shortest route depending on where it is. Our goal is to build an AI that will return that shortest route, wherever the

robot is. But then as we see, locations K and L are in the Top 3 priorities. Hence we will want to implement an option for our Autonomous Warehouse Robot to go by some intermediary locations before reaching its final top priority location.

The way the system computes the priorities of the locations is out of the scope of this case study. The reason for this is that there can be many ways, from simple rules or algorithms, to deterministic computations, to machine learning. But most of these ways would not be artificial intelligence as we know it today. What we really want to focus on is core AI, encompassing Q-Learning, Deep Q-Learning and other branches of Reinforcement Learning. So we will just say for example that Location G is top priority because one of the most loyal platinum customers of the company placed an urgent order of a product stored in location G which therefore must be delivered as soon as possible.

Therefore in conclusion, our mission is to build an AI that will always take the shortest route to the top priority location, whatever the location it starts from, and having the option to go by an intermediary location which is in the top 3 priorities.

1.1.2 Environment to define

When building an AI, the first thing we always have to do is to define the environment. And defining an environment always requires the three following elements:

- Defining the states
- Defining the actions
- Defining the rewards

Let's define these three elements, one by one.

Defining the states.

Let's start with the states. The input state is simply the location where our Autonomous Warehouse Robot is at each time t. However since we will build our AI with mathematical equations, we will encode the locations names (A, B, C,...) into index numbers, with respect to the following mapping:

Location	State
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	10
L	11

There is a specific reason why we encode the states with indexes from 0 to 11, instead of other integers. The reason is that we will work with matrices, a matrix of rewards and a matrix of Q-Values, and each line/column of these matrices will correspond to a specific location. For example, the first line of each matrix, which has index 0, corresponds to location A. The second line/column, which has index 1, corresponds to location B. Etc. We will see the purpose of working with matrices in more details a bit later.

Defining the actions.

Now let's define the possible actions to play. The actions are simply the next moves the robot can make to go from one location to the next. So for example, let's say the robot is in location J, the possible actions that the robot can play is to go to I, to F, or to K. And again, since we will work with mathematical equations, we will encode these actions with the same indexes as for the states. Hence, following our same example where the robot is in location J at a specific time, the possible actions that the robot can play are, according to our previous mapping above: 5, 8 and 10. Indeed, the index 5 corresponds to F, the index 8 corresponds to I, and the index 10 corresponds to K. Therefore eventually, the total list of actions that the AI can play overall is the following:

```
actions = [0,1,2,3,4,5,6,7,8,9,10,11]
```

Obviously, when being in a specific location, there are some actions that the robot cannot play. Taking the same previous example, if the robot is in location J, it can play the actions 5, 8 and 10, but it cannot play the other actions. We will make sure to specify that by attributing a 0 reward to the actions it cannot play, and a 1 reward to the actions it can play. And that brings us to the rewards.

Defining the rewards.

The last thing that we have to do now to build our environment is to define a system of rewards. More specifically, we have to define a reward function R that takes as inputs a state s and an action a, and returns a numerical reward that the AI will get by playing the action a in the state s:

$$R : (\text{state}, \text{action}) \mapsto r \in \mathbb{R}$$

So how are we going to build such a function for our case study? Here this is simple. Since there is a discrete and finite number of states (the indexes from 0 to 11), as well as a discrete and finite number of actions (same indexes from 0 to 11), the best way to build our reward function R is to simply make a matrix. Our reward function will exactly be a matrix of 12 rows and 12 columns, where the rows correspond to the states, and the columns correspond to the actions. That way, in our function "$R : (s, a) \mapsto r \in \mathbb{R}$", s will be the row index of the matrix, a will be the column index of the matrix, and r will be the cell of indexes (s, a) in the matrix.

Therefore the only thing that we have to do now to define our reward function is simply to populate this matrix with the numerical rewards. And as we just said in the previous paragraph, what we have to do first is to attribute, for each of the 12 locations, a 0 reward to the actions that the robot cannot play, and a 1 reward to the actions the robot can play. By doing that for each of the 12 locations, we will end up with a matrix of rewards. Let's build it step by step, starting with the first location: location A.

When being in location A, the robot can only go to location B. Therefore, since location A has index 0 (first row of the matrix) and location B has index 1 (second column of the matrix), the first row of the matrix of rewards will get a 1 on the second column, and a 0 on all the other columns, just like so:

Case Study #1 - Optimizing Warehouse Flows

Defining the Rewards:

	A	B	C	D	E	F	G	H	I	J	K	L
A	0	1	0	0	0	0	0	0	0	0	0	0
B												
C												
D												
E												
F												
G												
H												
I												
J												
K												
L												

Artificial Intelligence for Business © SuperDataScience

Now let's move on to location B. When being in location B, the robot can only go to three different locations: A, C and F. Since B has index 1 (second row), and A, C, F have respective indexes 0, 2, 5 (1st, 3rd, and 6th column), then the second row of the matrix of rewards will get a 1 on the 1st, 3rd and 6th columns, and 0 on all the other columns. Hence we get:

Case Study #1 - Optimizing Warehouse Flows

Defining the Rewards:

	A	B	C	D	E	F	G	H	I	J	K	L
A	0	1	0	0	0	0	0	0	0	0	0	0
B	1	0	1	0	0	1	0	0	0	0	0	0
C												
D												
E												
F												
G												
H												
I												
J												
K												
L												

Artificial Intelligence for Business © SuperDataScience

Then same, C (of index 2) is only connected to B and G (of indexes 1 and 6) so the third row of the matrix of rewards is:

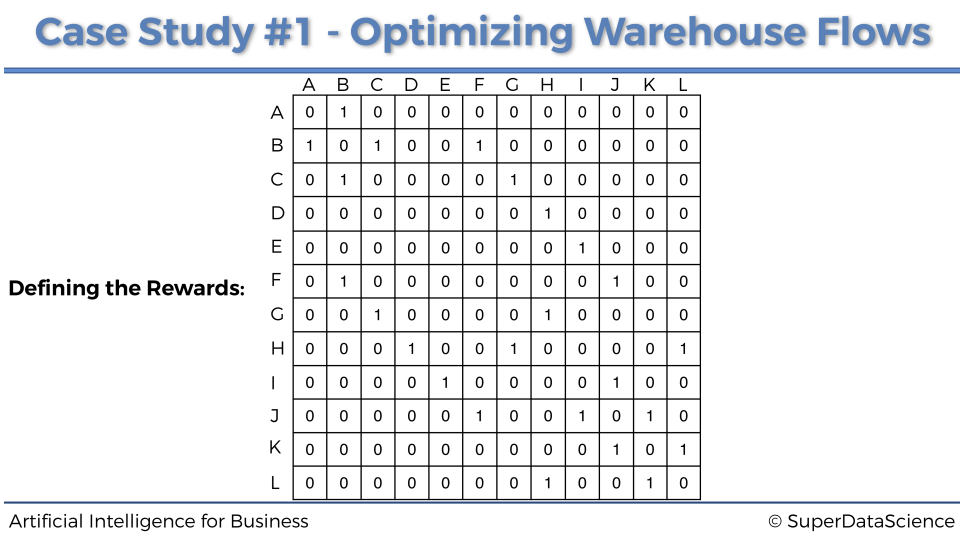
Case Study #1 - Optimizing Warehouse Flows

Defining the Rewards:

	A	B	C	D	E	F	G	H	I	J	K	L
A	0	1	0	0	0	0	0	0	0	0	0	0
B	1	0	1	0	0	1	0	0	0	0	0	0
C	0	1	0	0	0	0	1	0	0	0	0	0
D												
E												
F												
G												
H												
I												
J												
K												
L												

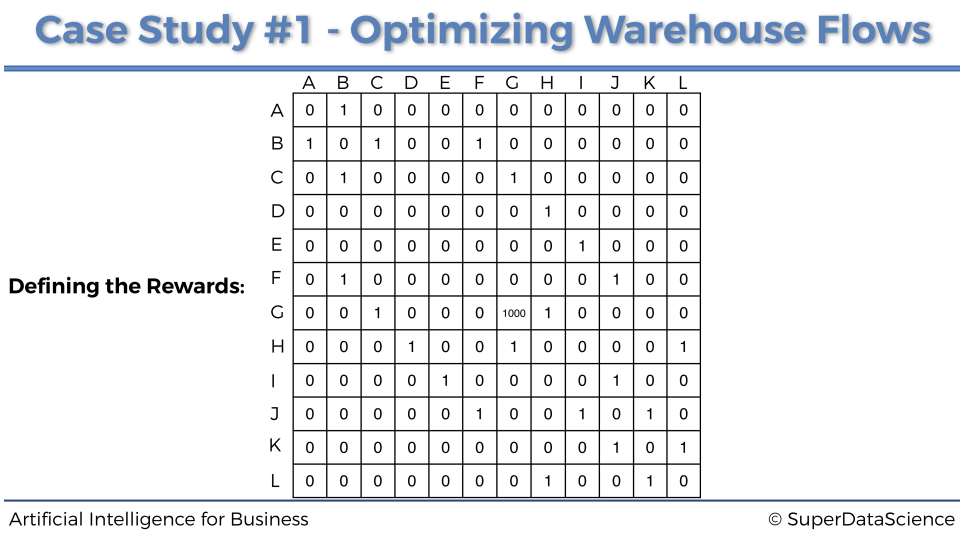
Artificial Intelligence for Business © SuperDataScience

By doing the same for all the other locations, we eventually get our final matrix of rewards:



Congratulations, we have just defined the rewards. We did it by simply building this matrix of rewards. It is important to understand that this is usually the way we define the system of rewards when doing Q-Learning with a finite number of inputs and actions. In Case Study 2, you will see that we will proceed very differently.

We are almost done, the only thing we need to do left is to attribute high rewards to the top priority locations. This will be done by the computer system that returns the priorities of product collection for each of the 12 locations. Therefore, since location G is the top priority, the computer system will update the matrix of rewards by attributing a high reward in the cell (G,G):



And that's how the system of rewards will work with Q-Learning. We attribute the highest reward (here 1000) to the top priority location G. Then you will see in the video lectures how we can attribute a lower high reward to the second top priority location (location K), to make our robot go by this intermediary top priority location, therefore optimizing the warehouse flows.

1.2 AI Solution

The AI Solution that will solve the problem described above is a Q-Learning model. Since the latter is based on Markov Decision Processes, or MDPs, we will start by explaining what they are, and then we will move on to the intuition and maths details behind the Q-Learning model.

1.2.1 Markov Decision Processes

A Markov Decision Process is a tuple (S, A, T, R) where:

- S is the set of the different states. Therefore in our case study:

$$S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

- A is the set of the different actions that can be played at each time t. Therefore in our case study:

$$A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

- T is called the transition rule:

$$T : (s_t \in S, s_{t+1} \in S, a_t \in A) \mapsto \mathbb{P}(s_{t+1}|s_t, a_t)$$

  where $\mathbb{P}(s_{t+1}|s_t, a_t)$ is the probability to reach the future state s_{t+1} when playing the action a_t in the state s_t. Therefore T is the probability distribution of the future states at time $t+1$ given the current state and the action played at time t. Accordingly, we can predict the future state s_{t+1} by taking a random draw from that distribution T:

$$s_{t+1} \sim T(s_t, ., a_t)$$

  In our case study, you will see through our implementation that this T distribution of our AI will simply be the uniform distribution, which is a classic choice of distribution working very well when doing Q-Learning.

- R is the reward function:

$$R : (s_t \in S, a_t \in A) \mapsto r_t \in \mathbb{R}$$

  where r_t is the reward obtained after playing the action a_t in the state s_t. In our case study, this reward function is exactly the matrix we defined previously.

After defining the MDP, it is now important to remind that it relies on the following assumption: the probability of the future state s_{t+1} only depends on the current state s_t and action a_t, and doesn't depend on any of the previous states and actions. That is:

$$\mathbb{P}(s_{t+1}|s_0, a_0, s_1, a_1, ..., s_t, a_t) = \mathbb{P}(s_{t+1}|s_t, a_t)$$

Hence in other words, a Markov Decision Process has no memory.

Now let's recap what is going on in terms of MDPs. At every time t:

1. The AI observes the current state s_t.
2. The AI plays the action a_t.
3. The AI receives the reward $r_t = R(s_t, a_t)$.
4. The AI enters the following state s_{t+1}.

So now the question is:

How does the AI know which action to play at each time t?

To answer this question, we need to introduce the policy function. The policy function π is exactly the function that, given a state s_t, returns the action a_t:

$$\pi : s_t \in S \mapsto a_t \in A$$

Let's denote by Π the set of all possible policy functions. Then the choice of the best actions to play becomes an optimization problem. Indeed, it comes down to finding the optimal policy π^* that maximizes the accumulated reward:

$$\pi^* = \underset{\pi \in \Pi}{\operatorname{argmax}} \sum_{t \geq 0} R(s_t, \pi(s_t))$$

Therefore of course the question becomes:

How to find this optimal policy π^* ?

This is where Q-Learning comes into play.

1.2.2 Q-Learning

Before we start getting into the details of Q-Learning, we need to explain the concept of the Q-Value.

The Q-Value

To each couple of state and action (s, a), we are going to associate a numeric value $Q(s, a)$:

$$Q : (s \in S, a \in A) \mapsto Q(s, a) \in \mathbb{R}$$

We will say that $Q(s, a)$ is "the Q-value of the action a played in the state s".

To understand the purpose of this "Q-Value", we need to introduce the Temporal Difference.

The Temporal Difference

At the beginning $t = 0$, all the Q-values are initialized to 0:

$$\forall s \in S, a \in A, Q(s, a) = 0$$

Now let's suppose we are at time t, in a certain state s_t. We play a random action a_t, which brings us to the state s_{t+1} and we get the reward $R(s_t, a_t)$.

We can now introduce the Temporal Difference, which is at the heart of Q-Learning. The Temporal Difference at time t, denoted by $TD_t(s_t, a_t)$, is the difference between:

- $R(s_t, a_t) + \gamma \max_a(Q(s_{t+1}, a))$, that is the reward $R(s_t, a_t)$ obtained by playing the action a_t in the state s_t, plus the Q-Value of the best action played in the future state s_{t+1}, discounted by a factor $\gamma \in [0, 1]$, called the discount factor.
- and $Q(s_t, a_t)$, that is the Q-Value of the action a_t played in the state s_t,

thus leading to:

$$TD_t(s_t, a_t) = R(s_t, a_t) + \gamma \max_a(Q(s_{t+1}, a)) - Q(s_t, a_t)$$

OK great, but what exactly is the purpose of this Temporal Difference $TD_t(s_t, a_t)$?

Let's answer this question to give us some better AI intuition. $TD_t(s_t, a_t)$ is like an intrinsic reward. The AI will learn the Q-values in such a way that:

- If $TD_t(s_t, a_t)$ is high, the AI gets a "good surprise".
- If $TD_t(s_t, a_t)$ is small, the AI gets a "frustration".

To that extent, the AI will iterate some updates of the Q-Values (through an equation called the Bellman equation) towards higher temporal differences.

Accordingly, in the final next step of the Q-Learning algorithm, we use the Temporal Difference to reinforce the couples (state, action) from time $t - 1$ to time t, according to the following equation:

$$Q_t(s_t, a_t) = Q_{t-1}(s_t, a_t) + \alpha TD_t(s_t, a_t)$$

where $\alpha \in \mathbb{R}$ is the learning rate, which dictates how fast the learning of the Q-Values goes, or how big the updates of the Q-Values are. Its value is usually a real number chosen between 0 and 1, like for example 0.01, 0.05, 0.1 or 0.5. The lower is its value, the smaller will be the updates of the Q-Values and the longer will be the Q-Learning. The higher is its value, the bigger will be the updates of the Q-Values and the faster will be the Q-Learning.

This equation above is the Bellman equation. It is the pillar of Q-Learning.

With this point of view, the Q-Values measure the accumulation of surprise or frustration associated with the couple of action and state (s_t, a_t). In the surprise case, the AI is reinforced, and in the frustration case, the AI is weakened. Hence we want to learn the Q-Values that will give the AI the maximum "good surprise".

Accordingly, the decision of which action to play mostly depends on the Q-value $Q(s_t, a_t)$. If the action a_t played in the state s_t is associated with a high Q-Value $Q(s_t, a_t)$, the AI will have a higher tendency to choose a_t. On the other hand if the action a_t played in the state s_t is associated with a small Q-value $Q(s_t, a_t)$, the AI will have a smaller tendency to choose a_t.

There are several ways of choosing the best action to play. First, when being in a certain state s_t, we could simply take the action a_t that maximizes the Q-Value $Q(s_t, a_t)$:

$$a_t = \underset{a}{\text{argmax}}(Q(s_t, a))$$

This solution is the Argmax method.

Another great solution, which turns out to be an even better solution for complex problems, is the Softmax method.

The Softmax method consists of considering for each state s the following distribution:

$$W_s : a \in A \mapsto \frac{\exp(Q(s,a))^\tau}{\sum_{a'} \exp(Q(s,a'))^\tau} \text{ with } \tau \geq 0$$

Then we choose which action a to play by taking a random draw from that distribution:

$$a \sim W_s(.)$$

However the problem we will solve in Case Study 1 will be simple enough to use the Argmax method, so this is what we will choose.

1.2.3 The whole Q-Learning algorithm

Let's summarize the different steps of the whole Q-Learning process:

Initialization:

For all couples of states s and actions a, the Q-Values are initialized to 0:

$$\forall s \in S, a \in A, Q_0(s,a) = 0$$

We start in the initial state s_0. We play a random possible action and we reach the first state s_1.

Then for each $t \geq 1$, we will repeat for a certain number of times (1000 times in our code) the following:

1. We select a random state s_t from our 12 possible states:

$$s_t = \text{random}(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)$$

2. We play a random action a_t that can lead to a next possible state, i.e., such that $R(s_t, a_t) > 0$:

$$a_t = \text{random}(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11) \text{ s.t. } R(s_t, a_t) > 0$$

3. We reach the next state s_{t+1} and we get the reward $R(s_t, a_t)$
4. We compute the Temporal Difference $TD_t(s_t, a_t)$:

$$TD_t(s_t, a_t) = R(s_t, a_t) + \gamma \underset{a}{\text{max}}(Q(s_{t+1}, a)) - Q(s_t, a_t)$$

5. We update the Q-value by applying the Bellman equation:

$$Q_t(s_t, a_t) = Q_{t-1}(s_t, a_t) + \alpha TD_t(s_t, a_t)$$

1.3 Q-Learning Implementation

Now let's provide and explain the whole implementation of this Q-Learning model, the solution of our warehouse flows optimization problem.

First, we start by importing the libraries that will be used in this implementation. These only include the numpy library, which offers a practical way of working with arrays and mathematical operations:

```
# Importing the libraries
import numpy as np
```

Then we set the parameters of our model. These include the discount factor γ and the learning rate α, which as we saw in Section 1.2, are the only parameters of the Q-Learning algorithm:

```
# Setting the parameters gamma and alpha for the Q-Learning
gamma = 0.75
alpha = 0.9
```

The two previous code sections were simply the introductory sections, before really starting to build our AI model. Now the next step is to start the first part of our implementation: Part 1 - Defining the Environment. And for that of course, we begin by defining the states, with a dictionary mapping the locations names (in letters from A to L) into the states (in indexes from 0 to 11):

```
# PART 1 - DEFINING THE ENVIRONMENT

# Defining the states
location_to_state = {'A': 0,
                     'B': 1,
                     'C': 2,
                     'D': 3,
                     'E': 4,
                     'F': 5,
                     'G': 6,
                     'H': 7,
                     'I': 8,
                     'J': 9,
                     'K': 10,
                     'L': 11}
```

Then we define the actions, with a simple list of indexes from 0 to 11. Remember that each action index corresponds to the next state (next location) where that action leads to:

```
# Defining the actions
actions = [0,1,2,3,4,5,6,7,8,9,10,11]
```

And eventually, we define the rewards, by creating a matrix of rewards, where the rows correspond to the current states s_t, the columns correspond to the actions a_t leading to the next state s_{t+1}, and the cells contain the rewards $R(s_t, a_t)$. If a cell (s_t, a_t) has a 1, that means that we can play the action a_t from the

current state s_t to reach the next state s_{t+1}. If a cell (s_t, a_t) has a 0, that means that we cannot play the action a_t from the current state s_t to reach any next state s_{t+1}. And for now we will manually put a high reward (1000) inside the cell corresponding to location G, because it is the top priority location where the autonomous warehouse has to go to collect the products. Since location G has encoded index state 6, we put a 1000 reward on the cell of row 6 and column 6. Then later on we will improve our solution by implementing an automatic way of going to the top priority location, without having to manually update the matrix of rewards and leaving it initialized with 0s and 1s just as it should be. But in the meantime, here is below our matrix of rewards including the manual update:

```
# Defining the rewards
R = np.array([[0,1,0,0,0,0,0,0,0,0,0,0],
              [1,0,1,0,0,1,0,0,0,0,0,0],
              [0,1,0,0,0,0,1,0,0,0,0,0],
              [0,0,0,0,0,0,0,1,0,0,0,0],
              [0,0,0,0,0,0,0,0,1,0,0,0],
              [0,1,0,0,0,0,0,0,0,1,0,0],
              [0,0,1,0,0,0,1000,1,0,0,0,0],
              [0,0,0,1,0,0,1,0,0,0,0,1],
              [0,0,0,0,1,0,0,0,0,1,0,0],
              [0,0,0,0,0,1,0,0,1,0,1,0],
              [0,0,0,0,0,0,0,0,0,1,0,1],
              [0,0,0,0,0,0,0,1,0,0,1,0]])
```

That closes this first part. Now let's begin the second part of our implementation: Part 2 - Building the AI Solution with Q-Learning. To that extent, we are going to follow the Q-Learning algorithm exactly as it was provided in Section 1.2. Hence we first initialize all the Q-Values, by creating our matrix of Q-Values full of zeros (in which same, the rows correspond to the current states s_t, the columns correspond to the actions a_t leading to the next state s_{t+1}, and the cells contain the Q-Values $Q(s_t, a_t)$):

```
# PART 2 - BUILDING THE AI SOLUTION WITH Q-LEARNING

# Initializing the Q-Values
Q = np.array(np.zeros([12,12]))
```

Then of course we implement the Q-Learning process, with a for loop over 1000 iterations, repeating 1000 times the steps of the Q-Learning process provided at the end of Section 1.2:

```
# Implementing the Q-Learning process
for i in range(1000):
    current_state = np.random.randint(0,12)
    playable_actions = []
    for j in range(12):
        if R[current_state, j] > 0:
            playable_actions.append(j)
    next_state = np.random.choice(playable_actions)
    TD = R[current_state, next_state] + gamma*Q[next_state, np.argmax(Q[next_state,])]
         - Q[current_state, next_state]
    Q[current_state, next_state] = Q[current_state, next_state] + alpha*TD
```

Optional: at this stage of the code, our matrix of Q-Values is ready. We can have a look at it by executing the whole code we have implemented so far, and by entering the following print in the console:

```
print("Q-Values:")
print(Q.astype(int))
```

And we obtain the following matrix of final Q-Values:

```
In [2]: print("Q-Values:")
   ...: print(Q.astype(int))
Q-Values:
[[   0 1661    0    0    0    0    0    0    0    0    0    0]
 [1246    0 2213    0    0 1246    0    0    0    0    0    0]
 [   0 1661    0    0    0    0 2970    0    0    0    0    0]
 [   0    0    0    0    0    0    0 2225    0    0    0    0]
 [   0    0    0    0    0    0    0    0  703    0    0    0]
 [   0 1661    0    0    0    0    0    0    0  931    0    0]
 [   0    0 2213    0    0    0 3968 2225    0    0    0    0]
 [   0    0    0 1661    0    0 2968    0    0    0    0 1670]
 [   0    0    0    0  528    0    0    0    0  936    0    0]
 [   0    0    0    0    0 1246    0    0  703    0 1246    0]
 [   0    0    0    0    0    0    0    0    0  936    0 1661]
 [   0    0    0    0    0    0    0 2225    0    0 1246    0]]
```

For more visual clarity, you can even check the matrix of Q-Values directly in Variable Explorer, by double clicking on Q. Then to get the Q-Values as integers you can click on "Format" and inside enter a float formatting of "%.0f". You will obtain this, which is a bit more clear since you can see the indexes of the rows and columns in your matrix Q:

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1661	0	0	0	0	0	0	0	0	0	0
1	1247	0	2214	0	0	1247	0	0	0	0	0	0
2	0	1661	0	0	0	0	2970	0	0	0	0	0
3	0	0	0	0	0	0	0	2226	0	0	0	0
4	0	0	0	0	0	0	0	0	703	0	0	0
5	0	1661	0	0	0	0	0	0	0	931	0	0
6	0	0	2214	0	0	0	3968	2226	0	0	0	0
7	0	0	0	1661	0	0	2968	0	0	0	0	1670
8	0	0	0	0	528	0	0	0	0	936	0	0
9	0	0	0	0	0	1247	0	0	703	0	1246	0
10	0	0	0	0	0	0	0	0	0	936	0	1661
11	0	0	0	0	0	0	0	2226	0	0	1247	0

Good, now that we have our matrix of Q-Values, we are ready to go into production! Hence we can move on to the third part of the implementation, Part 3 - Going into Production, inside which we will compute the optimal path from any starting location to any ending top priority location. The idea here will be to implement a "route" function, that will take as inputs the starting location where our autonomous warehouse robot is located at a specific time and the ending location where it has to go in top priority, and that will return as output the shortest route inside a list. However since we want to input the locations with there names (in letters), as opposed to their states (in indexes), we will need a dictionary that maps the locations states (in indexes) to the locations names (in letters). And that is the first thing we will do here in this third part, using a trick to inverse our previous dictionary "location_to_state", since indeed we simply want to get the exact inverse mapping from this dictionary:

```
# PART 3 - GOING INTO PRODUCTION

# Making a mapping from the states to the locations
state_to_location = {state: location for location, state in location_to_state.items()}
```

This is when the most important code section comes into play. We are about to implement the final "route()" function that will take as inputs the starting and ending locations, and that will return the optimal path between these two locations. To explain exactly what this route function will do, let's enumerate the different steps of the process, when going from location E to location G:

1. We start at our starting location E.
2. We get the state of location E, which according to our location_to_state mapping is $s_0 = 4$.
3. On the row of index $s_0 = 4$ in our matrix of Q-Values, we find the column that has the maximum Q-Value (703).
4. This column has index 8, so we play the action of index 8 which leads us to the next state $s_{t+1} = 8$.
5. We get the location of state 8, which according to our state_to_location mapping is location I. Hence our next location is location I, which is appended to our list containing the optimal path.
6. We repeat the same previous 5-steps from our new starting location I, until we reach our final destination, location G.

Hence, since we don't know how many locations we will have to go through between the starting and ending locations, we have to make a while loop that will repeat the 5-steps process described above, and that will stop as soon as we reach the ending top priority location:

```
# Making the final function that will return the optimal route
def route(starting_location, ending_location):
    route = [starting_location]
    next_location = starting_location
    while (next_location != ending_location):
        starting_state = location_to_state[starting_location]
        next_state = np.argmax(Q[starting_state,])
        next_location = state_to_location[next_state]
        route.append(next_location)
        starting_location = next_location
    return route
```

Congratulations, our tool is now ready! When we test it to go from E to G, we get indeed the two possible optimal paths after printing the final route executing the whole code several times:

```
# Printing the final route
print('Route:')
route('E', 'G')

Route:
Out[1]: ['E', 'I', 'J', 'F', 'B', 'C', 'G']
Out[2]: ['E', 'I', 'J', 'K', 'L', 'H', 'G']
```

Good, we have a first version of the model that is well functioning. But we can improve it in two ways. First, by automating the reward attribution to the top priority location, so that we don't have to do it manually. And second, by adding a feature that gives us the option to go by an intermediary location before going to the top priority location. That intermediary location should be of course in the Top 3 priority locations. And as a matter of fact, in our top priority locations ranking, the second top priority location is location K. Therefore, in order to optimize even more the warehouse flows, our autonomous warehouse robot must go by location K to collect the products on its way to the top priority location G. A way to do this is to have the option to go by any intermediary location in the process of our "route()" function. And this is exactly what we will implement as a second improvement. But first, let's implement the first improvement, that automates the reward attribution.

The way to do that is two folds: first we must make a copy (called R_new) of our reward matrix inside which the route() function will automatically update the reward in the cell of the ending location. Indeed, the ending location is one of the inputs of the route() function, so using our location_to_state dictionary we can very easily find that cell and update its reward to 1000. And second, we must include the whole Q-Learning algorithm (including the initialization step) inside the route function, right after we make that update of the reward in our copy of the rewards matrix. Indeed, in our previous implementation above, the Q-Learning process happens on the original version of the rewards matrix, which is now supposed to stay as it is, i.e. initialized to 1s and 0s only. Therefore we must include the Q-Learning process inside the route function, and make it happen on our copy R_new of the rewards matrix, instead of the original rewards matrix R. Hence, our full implementation becomes the following:

```
# Artificial Intelligence for Business
# Optimizing Warehouse Flows with Q-Learning

# Importing the libraries
import numpy as np

# Setting the parameters gamma and alpha for the Q-Learning
gamma = 0.75
alpha = 0.9

# PART 1 - DEFINING THE ENVIRONMENT

# Defining the states
location_to_state = {'A': 0,
                     'B': 1,
                     'C': 2,
                     'D': 3,
                     'E': 4,
                     'F': 5,
                     'G': 6,
                     'H': 7,
                     'I': 8,
                     'J': 9,
                     'K': 10,
                     'L': 11}

# Defining the actions
actions = [0,1,2,3,4,5,6,7,8,9,10,11]
```

```
# Defining the rewards
R = np.array([[0,1,0,0,0,0,0,0,0,0,0,0],
              [1,0,1,0,0,1,0,0,0,0,0,0],
              [0,1,0,0,0,0,1,0,0,0,0,0],
              [0,0,0,0,0,0,0,1,0,0,0,0],
              [0,0,0,0,0,0,0,0,1,0,0,0],
              [0,1,0,0,0,0,0,0,0,1,0,0],
              [0,0,1,0,0,0,1,1,0,0,0,0],
              [0,0,0,1,0,0,1,0,0,0,0,1],
              [0,0,0,0,1,0,0,0,0,1,0,0],
              [0,0,0,0,0,1,0,0,1,0,1,0],
              [0,0,0,0,0,0,0,0,0,1,0,1],
              [0,0,0,0,0,0,0,1,0,0,1,0]])

# PART 2 - BUILDING THE AI SOLUTION WITH Q-LEARNING

# Making a mapping from the states to the locations
state_to_location = {state: location for location, state in location_to_state.items()}

# Making the final function that will return the route
def route(starting_location, ending_location):
    R_new = np.copy(R)
    ending_state = location_to_state[ending_location]
    R_new[ending_state, ending_state] = 1000
    Q = np.array(np.zeros([12,12]))
    for i in range(1000):
        current_state = np.random.randint(0,12)
        playable_actions = []
        for j in range(12):
            if R_new[current_state, j] > 0:
                playable_actions.append(j)
        next_state = np.random.choice(playable_actions)
        TD = R_new[current_state, next_state]
             + gamma * Q[next_state, np.argmax(Q[next_state,])]
             - Q[current_state, next_state]
        Q[current_state, next_state] = Q[current_state, next_state] + alpha * TD
    route = [starting_location]
    next_location = starting_location
    while (next_location != ending_location):
        starting_state = location_to_state[starting_location]
        next_state = np.argmax(Q[starting_state,])
        next_location = state_to_location[next_state]
        route.append(next_location)
        starting_location = next_location
    return route

# PART 3 - GOING INTO PRODUCTION

# Printing the final route
print('Route:')
route('E', 'G')
```

By executing this new code several times, we get of course the same two possible optimal paths as before.

Now let's tackle the second improvement. There are three ways to add the option of going by the intermediary location K, the second top priority location:

1. We give a high reward to the action leading from location J to location K. This high reward has to be larger than 1, and below 1000. Indeed it has to be larger than 1 so that the Q-Learning process favors the action leading from J to K, as opposed to the action leading from J to F which has reward 1. And it must be below than 1000 so we have to keep the highest reward on the top priority location, to make sure we end up there. Hence for example, in our rewards matrix we can give a high reward of 500 to the cell in the row of index 9 and the column of index 10, since indeed that cell corresponds to the action leading from location J (state index 9) to location K (state index 10). That way our autonomous warehouse robot will always go by location K on its way to location G. Here is how the matrix of rewards would be in that case:

```
# Defining the rewards
R = np.array([[0,1,0,0,0,0,0,0,0,0,0,0],
              [1,0,1,0,0,1,0,0,0,0,0,0],
              [0,1,0,0,0,0,1,0,0,0,0,0],
              [0,0,0,0,0,0,0,1,0,0,0,0],
              [0,0,0,0,0,0,0,0,1,0,0,0],
              [0,1,0,0,0,0,0,0,0,1,0,0],
              [0,0,1,0,0,0,1,1,0,0,0,0],
              [0,0,0,1,0,0,1,0,0,0,0,1],
              [0,0,0,0,1,0,0,0,0,1,0,0],
              [0,0,0,0,0,1,0,0,1,0,500,0],
              [0,0,0,0,0,0,0,0,0,0,1,0,1],
              [0,0,0,0,0,0,0,1,0,0,1,0]])
```

2. We give a bad reward to the action leading from location J to location F. This bad reward just has to be below 0. Indeed by punishing this action with a bad reward the Q-Learning process will never favor that action leading from J to F. Hence for example, in our rewards matrix we can give a bad reward of -500 to the cell in the row of index 9 and the column of index 5, since indeed that cell corresponds to the action leading from location J (state index 9) to location F (state index 5). That way our autonomous warehouse robot will never go trough location F on its way to location G. Here is how the matrix of rewards would be in that case:

```
# Defining the rewards
R = np.array([[0,1,0,0,0,0,0,0,0,0,0,0],
              [1,0,1,0,0,1,0,0,0,0,0,0],
              [0,1,0,0,0,0,1,0,0,0,0,0],
              [0,0,0,0,0,0,0,1,0,0,0,0],
              [0,0,0,0,0,0,0,0,1,0,0,0],
              [0,1,0,0,0,0,0,0,0,1,0,0],
              [0,0,1,0,0,0,1,1,0,0,0,0],
              [0,0,0,1,0,0,1,0,0,0,0,1],
              [0,0,0,0,1,0,0,0,0,1,0,0],
              [0,0,0,0,0,-500,0,0,1,0,1,0],
              [0,0,0,0,0,0,0,0,0,0,1,0,1],
              [0,0,0,0,0,0,0,1,0,0,1,0]])
```

3. We make an additional best_route() function, taking as inputs the three starting, intermediary and ending locations, that will call our previous route() function twice, a first time from the starting location to the intermediary location, and a second time from the intermediary location to the ending location.

The first two ideas are easy to implement manually, but very tricky to implement automatically. Indeed, it is easy to find automatically the index of the intermediary location where we want to go by, but very difficult to get the index of the location that leads to that intermediary location, since it depends on the starting location and the ending location. You can try to implement either the first or second idea, you will see what I mean. Accordingly, we will implement the third idea, which can be coded in just two extra lines of code:

```
# Making the final function that returns the optimal route
def best_route(starting_location, intermediary_location, ending_location):
    return route(starting_location, intermediary_location)
           + route(intermediary_location, ending_location)[1:]
```

Eventually, the final code including that major improvement for our warehouse flows optimization, becomes:

```
# Artificial Intelligence for Business
# Optimizing Warehouse Flows with Q-Learning

# Importing the libraries
import numpy as np

# Setting the parameters gamma and alpha for the Q-Learning
gamma = 0.75
alpha = 0.9

# PART 1 - DEFINING THE ENVIRONMENT

# Defining the states
location_to_state = {'A': 0,
                     'B': 1,
                     'C': 2,
                     'D': 3,
                     'E': 4,
                     'F': 5,
                     'G': 6,
                     'H': 7,
                     'I': 8,
                     'J': 9,
                     'K': 10,
                     'L': 11}

# Defining the actions
actions = [0,1,2,3,4,5,6,7,8,9,10,11]

# Defining the rewards
R = np.array([[0,1,0,0,0,0,0,0,0,0,0,0],
              [1,0,1,0,0,1,0,0,0,0,0,0],
              [0,1,0,0,0,0,1,0,0,0,0,0],
              [0,0,0,0,0,0,0,1,0,0,0,0],
              [0,0,0,0,0,0,0,0,1,0,0,0],
              [0,1,0,0,0,0,0,0,0,1,0,0],
              [0,0,1,0,0,0,1,1,0,0,0,0],
              [0,0,0,1,0,0,1,0,0,0,0,1],
```

```
                [0,0,0,0,1,0,0,0,0,1,0,0],
                [0,0,0,0,0,1,0,0,1,0,1,0],
                [0,0,0,0,0,0,0,0,0,1,0,1],
                [0,0,0,0,0,0,0,1,0,0,1,0]])

# PART 2 - BUILDING THE AI SOLUTION WITH Q-LEARNING

# Making a mapping from the states to the locations
state_to_location = {state: location for location, state in location_to_state.items()}

# Making a function that returns the shortest route from a starting to ending location
def route(starting_location, ending_location):
    R_new = np.copy(R)
    ending_state = location_to_state[ending_location]
    R_new[ending_state, ending_state] = 1000
    Q = np.array(np.zeros([12,12]))
    for i in range(1000):
        current_state = np.random.randint(0,12)
        playable_actions = []
        for j in range(12):
            if R_new[current_state, j] > 0:
                playable_actions.append(j)
        next_state = np.random.choice(playable_actions)
        TD = R_new[current_state, next_state]
             + gamma * Q[next_state, np.argmax(Q[next_state,])]
             - Q[current_state, next_state]
        Q[current_state, next_state] = Q[current_state, next_state] + alpha * TD
    route = [starting_location]
    next_location = starting_location
    while (next_location != ending_location):
        starting_state = location_to_state[starting_location]
        next_state = np.argmax(Q[starting_state,])
        next_location = state_to_location[next_state]
        route.append(next_location)
        starting_location = next_location
    return route

# PART 3 - GOING INTO PRODUCTION

# Making the final function that returns the optimal route
def best_route(starting_location, intermediary_location, ending_location):
    return route(starting_location, intermediary_location)
           + route(intermediary_location, ending_location)[1:]

# Printing the final route
print('Route:')
best_route('E', 'K', 'G')
```

By executing this whole new code as many times as we want, we will always get the same expected output:

```
Best Route:
Out[1]: ['E', 'I', 'J', 'K', 'L', 'H', 'G']
```