**University of Maryland Global Campus**

DATA 640 – Predictive Modeling

**Assignment №5b – Reinforcement Learning main assignment**

**Deadline:** End of week 11, 11:59 pm Eastern Time

Submission via LEO.

*This is an* ***individual*** *assignment. Each student will complete the assignment outlined below and post his/her written results to the appropriate assignment. Please note that only 1 document is allowed to be submitted.*

Contents

[Grading criteria 1](#_Toc58548932)

[Reinforcement Learning Assignment Overview 1](#_Toc58548933)

[Detailed assignment instructions 2](#_Toc58548934)

[Submission 3](#_Toc58548935)

[Content 3](#_Toc58548936)

[Appendix 1: An introduction to Q-Learning: Reinforcement Learning by Sayak Paul 7](#_Toc58548937)

[Appendix2: Q-Learning using numpy Code by Sayak Paul 39](#_Toc58548938)

# **Grading criteria**

Submitted assignments will be graded for (a) content, (b) document quality (i.e., formatting, following guidelines, pleasant to read, etc.), and timeliness of submission. Assignments submitted late will be deducted 5 points for each day it is late.

# **Reinforcement Learning Assignment Overview**

You’ll learn basic concepts of Q-Learning and understand how they work in practice.

1. Complete the reinforcement learning practice exercise on Tic-Tac-Toe based on the article by Ryan Rudes.
2. Read the background material on reinforcement learning by Sayak Paul: [https://blog.floydhub.com/an-introduction-to-q-learning-reinforcement-learning/](about:blank) This article is reproduced in **Appendix 1** in this document.
3. Make sure you have an active Google account, and can access Google Colab: [https://colab.research.google.com](about:blank)
4. Follow the **Detailed assignment instructions**
5. Write the report, as specified in the **Content** section.

**NOTE: Be sure to allow appropriate time for working on this assignment.**

## **Detailed assignment instructions**

1. Log into to **Google Colab** and follow one of these two choices:
   1. File->New Notebook and copy and paste there the code in **Appendix 2**
   2. Or upload Notebook and select the file **Q-Learning using numpy.ipynb**
2. Make sure that you can run the notebook or the single block of code.
   1. The code in the appendix does not include some pictures and diagrams that are there for informational purposes in the notebook—but they both contain the same Python code.
   2. If you are running the notebook you should run all of the subsections in order.
   3. The code in the appendix will run all in one shot.
3. If you are successful, the result of print(get\_optimal\_route('L9', 'L1')) will be ['L9', 'L8', 'L5', 'L2', 'L1']. Explain why this is the case.
4. Review the code to make sure you understand it. In the writeup, you will need to provide an overview of the code using your own words. Do not copy the description from Sayak Paul’s article.
5. Run with different levels of hyperparameters gamma and alpha. What are these parameters for? For example, try 0.05 for both. What happens now? Write your findings and explanation. Remember to set back **gamma = 0.9** **and alpha = 0.75** after this before you proceed with the next questions.
6. When you run **print(get\_optimal\_route('L9', 'L1'))** how many times does the while loop in the **get\_optimal\_route()** get executed and why it is this number? Hint: one way to check is to add **steps =0** before the while loop. Then in the while loop insert **steps = steps +1** and then change **return route** to **return route, steps**
7. The Q-learning process for loop in get\_optimal\_route() performs 1000 iterations. Do you think these many iterations are required? Try with 50. Try with 200. Explain what happens and why you think it is happening. Remember to set it back to 1000 iterations before you work on the next questions.
8. Now try to run the reverse path print(get\_optimal\_route('L1', 'L9'))
   1. The code will most likely keep iterating without returning. Why do you think this is? Hint: Look at the rewards for L2. Can you get to L5 from L2?
   2. Make the change so that this case works; see the previous hint. And show the result you obtain.
9. The existing code has 9 states L1-L9. Add a tenth state L10, where it is only possible to go from L10 to L9 and vice versa as depicted in Figure 1.
   1. Make all of the required changes to the code.
   2. Run **print(get\_optimal\_route('L10', 'L1'))** and show your results and explain if they are as expected or not.
   3. Run **print(get\_optimal\_route('L10', 'L4'))** and show your results and explain if they are as expected or not.

L1

L4

L7

L3

L2

L5

L8

L6

L9

L10

Figure 1 Problem with 10 States

# **Submission**

Each student will submit a single document conforming to the guidelines and standards outlined below.

**Document format**:

* limited to 5 pages (**excluding** *title page*, *references*, and *appendix*),
* **Double-spaced**, 12-point Times New Roman font, 1” margins, Bottom-right page numbering.

**Note:** Submitted report must be either in MS Word or PDF format and titled: “*Assignmen5\_LastName*”.

Only one document will be allowed to be submitted.

# **Content**

(Note that the document must have at ***clearly marked sections*** at least for the items listed below)

1. **Title page** (*1 page limit*): course number and term, assignment number and project title, student name and contact information, instructor’s name. *Format it so it looks pleasant and presentable.* Follow formatting guidelines above.
2. **Introduction.** Provide a brief outline of this assignment.
3. **Overview of the Q-Learning Code.** Using your own words, provide a detailed overview of the code. This corresponds to question 4 of the **Detailed assignment instructions.**
4. **Answer questions 3, 5,6,7,8 and 9 of the Detailed assignment instructions** providing evidence with screenshots. Label each question separately.
5. **Conclusions and takeaways**. Glean any insights you found by doing this assignment.
6. **References** (*1 page limit*): List all references in APA format used in preparing this report. It is strongly recommended to use outside knowledge in setting-up the analysis or discussing the results where possible.
7. **Code Appendix:** Include the code you used for question 9 as an Appendix.

**University of Maryland Global Campus**

DATA 640 – Predictive Modeling

**Student Name**: **Assignment №5: Reinforcement Learning** Total points: 100

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Content** | **Explanation** | **Points** | | **Comment** |
| **Total** | **Earned** |
| **Written Report** |  |  |  |  |
| * 1. Intro-duction | **How well is the problem described and outlined?** | 5 |  |  |
| * 1. Question 4 | **Describe the Q-Leaning code in detail.** | 15 |  |  |
| * 1. Question 3 | If you are successful running the code, the result of print(get\_optimal\_route('L9', 'L1')) will be ['L9', 'L8', 'L5', 'L2', 'L1']. Explain why this is the case. | 15 |  |  |
| * 1. Question 5 | Run with different levels of hyperparameters gamma and alpha **Discuss the key results** | 10 |  |  |
| * + - 1. Question 6 | When you run **print(get\_optimal\_route('L9', 'L1'))** how many times does the while loop in the **get\_optimal\_route()** get executed? Why it is this number? | 10 |  |  |
| * + - 1. Question 7 | The Q-learning process for loop in get\_optimal\_route() performs 1000 iterations. Do you think these many iterations are required? **Try with 50. Try with 200.** Explain what happens and why you think it is happening. | 10 |  |  |
| * + - 1. Question 8 | Now try to run the reverse path print(get\_optimal\_route('L1', 'L9'))   * + The code will most likely keep iterating without returning. Why do you think this is? Hint: Look at the rewards for L2. Can it get to L5 from L2?   + Make the change so that this case works; see the previous hint. Show the results you obtain. | 15 |  |  |
| * + - 1. Question 9 | The existing code has 9 states L1-L9. Add a tenth state L10, where it is only possible to go from L10 to L9 and vice versa as depicted in Figure 1.   * + Make all of the required changes to the code.   + Run **print(get\_optimal\_route('L10', 'L1'))** and show your results and explain if they are as expected or not.   + Run **print(get\_optimal\_route('L10', 'L4'))** and show your results and explain if they are as expected or not. | 15 |  |  |
| * + - 1. Mechanics (spelling, grammar) | Is the **paper free of grammatical errors and spelling and punctuation?** Is the paper properly formatted? Does the paper flow well? How well does the paper “tell the story”? | 2 |  |  |
| * + - 1. Citations and References | Are all references and citations correctly written and presented? | 1 |  |  |
| * + - 1. **Conclusions and takeaway** | **Include insightful and meaningful conclusions, suggestions, and insights.** | 2 |  |  |
| **Total** |  | **100** |  |  |
|  |  |  |  |  |
| * + - 1. **Less** |  |  |  |  |
| Page limit | Is the submission written within specified page limits? 2.5 points will be deducted for submissions substantially over the page limits | - 2.5 |  |  |
| Late submission (less) | 2.5 points will be deducted for each day the assignment is late. | -2.5 each day |  |  |
| **Final Grade for Assignment 5** | Final grade equals:  .05 (5%) times the Tic-Tac-Toe assignment  .95 (95%) times the main assignment |  |  |  |

# **Appendix 1: An introduction to Q-Learning: Reinforcement Learning by Sayak Paul**

Source: <https://blog.floydhub.com/an-introduction-to-q-learning-reinforcement-learning/>

An introduction to Q-Learning: Reinforcement Learning



Have you ever trained a pet and rewarded it for every correct command you asked for? Do you know that this simple way of rewarding behavior can be modeled in a robot or a software program to make it do useful things? In this article, we are going to step into the world of **reinforcement learning**, another beautiful branch of artificial intelligence, which lets machines learn on their own in a way different from traditional machine learning. Particularly, we will be covering the simplest reinforcement learning algorithm i.e. the Q-Learning algorithm in great detail.

In the first half of the article, we will be discussing reinforcement learning in general with examples where reinforcement learning is not just desired but also required. We will then study the Q-Learning algorithm along with an implementation in Python using Numpy.

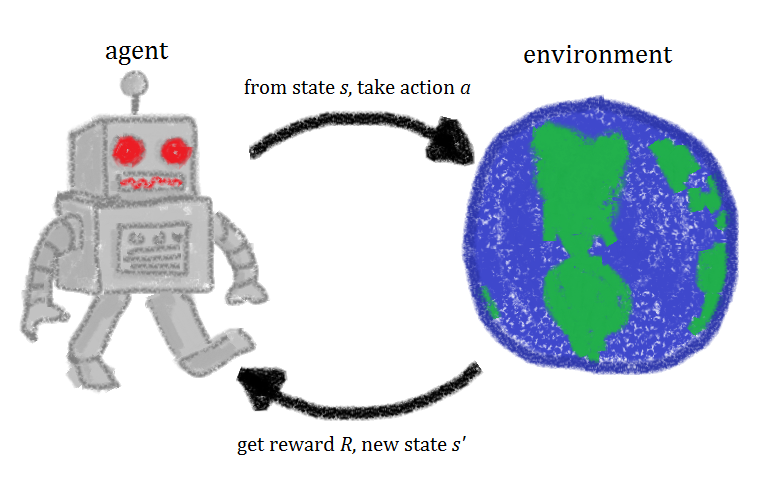
Let’s start, shall we?

Ready to build, train, and deploy AI?

[Get started with FloydHub's collaborative AI platform for free](https://www.floydhub.com/)

Try FloydHub for free

The brass tacks: What is reinforcement learning?

**A rough framework of reinforcement learning**

Throughout our lives, we perform a number of actions to pursue our dreams. Some of them bring us good rewards and others do not. Along the way, we keep exploring different paths and try to figure out which action might lead to better rewards. We work hard towards our dreams utilizing the feedback we get based on our actions to improve our strategies. They help us determine how close we are to achieving our goals. Our mental states change continuously to representing this closeness.

In that description of how we pursue our goals in daily life, we framed for ourselves a representative analogy of reinforcement learning. Let me summarize the above example reformatting the main points of interest.

* Our reality contains **environments** in which we perform numerous actions. Sometimes we get good or positive **rewards**for some of these **actions in order to achieve goals**. During the entire course of life, our mental and physical states evolve. We **strengthen** our actions in order to get as many rewards as possible.
* The key entities of interest are **environment**, **action**, **reward**and **state**. Let’s give ourselves a name as well - we are **agents**in this whole game of life.
* This whole paradigm of exploring our lives and learning from it through actions, rewards and states establishes the foundation of reinforcement learning. In fact, this is almost how we act in any given circumstance in our lives, isn’t it?

It turns out that the whole idea of reinforcement learning is pretty empirical in nature. Now, think for a moment, how would we train robots and machines to do the kind of useful tasks we humans do. Be it switching off the television, or moving things around, or organizing bookshelves. Fundamentally, these tasks are not about finding a function mapping inputs to outputs or finding hidden representations within input data. These are a completely different set of tasks and require a different learning paradigm for a computer to be able to perform these tasks.

[**Andriy Burkov** in his **The Hundred Page Machine Learning Book**](about:blank#hundred-page-machine-learning) describes reinforcement learning as:

**Reinforcement learning solves a particular kind of problem where decision making is sequential, and the goal is long-term, such as game playing, robotics, resource management, or logistics.**

For a robot, an environment is a place where it has been put to use. Remember this robot is itself the agent. For example, a textile factory where a robot is used to move materials from one place to another. We will come to its states, actions, rewards later. The tasks we discussed just now, have a property in common ---- **these tasks involve an environment and expect the agents to learn from that environment.** This is where traditional machine learning fails and hence the need for reinforcement learning.

So, the headline [AI Bots Join Forces To Beat Top Human Dota 2 Team](https://www.forbes.com/sites/samshead/2018/08/06/elon-musk-openai-bots-dota-2/?sh=6d5b7048494c) that shook the gaming world is a direct byproduct of reinforcement learning. Beyond beating humans in game-playing, there are some other marvelous use cases of reinforcement learning:

[Optimizing business processes](https://www.technologyreview.com/2016/03/18/161519/this-factory-robot-learns-a-new-job-overnight/)

[Minimizing energy costs](https://deepmind.com/blog/article/deepmind-ai-reduces-google-data-centre-cooling-bill-40)

[Maximizing revenue shares of a company](https://arxiv.org/pdf/1803.09967.pdf)

and lots [more](https://www.oreilly.com/radar/practical-applications-of-reinforcement-learning-in-industry/).

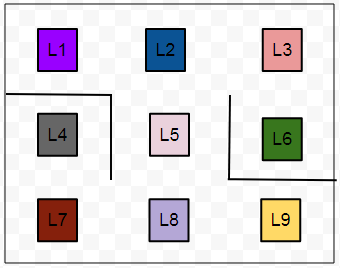
In the next section, we will begin by defining the problem which is very relevant in the field of reinforcement learning. After that, we will study its agents, environment, states, actions and rewards. We will then directly proceed towards the Q-Learning algorithm.

**Recipes for reinforcement learning**

It is good to have an established overview of the problem that is to be solved using reinforcement learning, Q-Learning in this case. It helps to define the main components of a reinforcement learning solution i.e., agents, environment, actions, rewards and states.

**Defining the problem statement**

We are to build a few autonomous robots for a guitar building factory. These robots will help the guitar luthiers by conveying them the necessary guitar parts that they would need in order to craft a guitar. These different parts are located at **nine** different positions within the factory warehouse. Guitars parts include polished wood stick for the fretboard, polished wood for the guitar body, guitar pickups and so on. The luthiers have prioritized the location that contains body woods to be the topmost. They have provided the priorities for other locations as well which we will look in a moment. These locations within the factory warehouse look like -

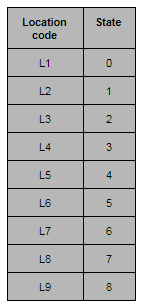
**A sample environment**

As we can see there are little obstacles present (represented with smoothed lines) in between the locations. **L6**is the top-priority location that contains the polished wood for preparing guitar bodies. Now, the task is **to enable the robots so that they can find the shortest route from any given location to another location on their own.**

The **agents**, in this case, are the robots. The **environment**is the guitar factory warehouse.

**The states**

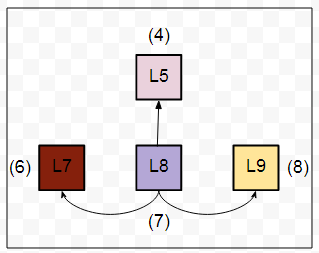
The states are the locations. The location in which a particular robot is present in the particular instance of time will denote its state. Machines understand numbers rather than letters. So, let’s map the location codes to numbers.



**Locations mapped as states**

**The actions**

In our example, the actions will be the direct locations that a robot can go to from a particular location. Consider, a robot is at the **L8** location and the direct locations to which it can move are **L5**, **L7**and **L9**. The below figure may come in handy in order to visualize this.

**Sample actions**

As you might have already guessed the set of actions here is nothing but the set of all possible states of the robot. For each location, the set of actions that a robot can take will be different. For example, the set of actions will change if the robot is in L1.

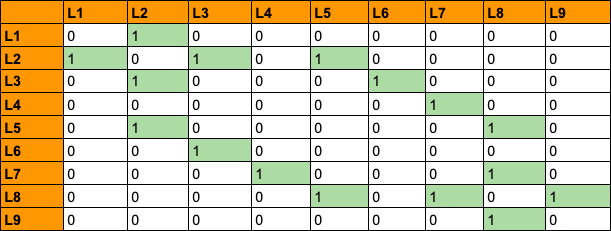
**The rewards**

**By now, we have the following two sets:**

A set of states: S=0,1,2,3,4,5,6,7,8S=0,1,2,3,4,5,6,7,8

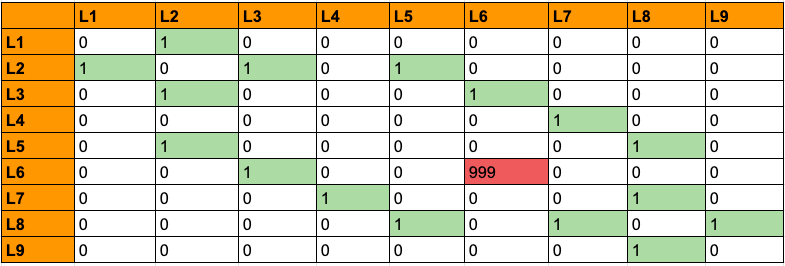
A set of actions: A=0,1,2,3,4,5,6,7,8A=0,1,2,3,4,5,6,7,8

* The rewards, now, will be given to a robot if a location (read it **state**) is directly reachable from a particular location. Let’s take an example:L9 is directly reachable from L8. So, if a robot goes from L8 to L9 and vice-versa, it will be rewarded by **1**.
* If a location is not directly reachable from a particular location, we do not give any reward (a reward of **0**). Yes, the reward is just a number here and nothing else. It enables the robots to make sense of their movements helping them in deciding what locations are directly reachable and what are not. With this cue, we can construct a reward table which contains all the reward values mapping between all the possible states (locations).

**Table of rewards**

In the above table, we have all the possible rewards that a robot can get by moving in between the different states.

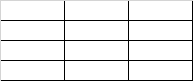
Now comes an interesting decision. Remember from the previous sections that the luthier prioritized L6 to be of the topmost? So, how do we incorporate this fact in the above table? This is done by associating the topmost priority location with a very higher reward than the usual ones. Let’s put **999**in the cell (**L6**, **L6**):

**Table of rewards with a higher reward for the topmost location**

We have now formally defined all the vital components for the solution we are aiming for the problem discussed above. We will shift gears a bit and study some of the fundamental concepts that prevail in the world of reinforcement learning. We will start with the [**Bellman Equation.**](https://www.youtube.com/watch?v=14BfO5lMiuk)

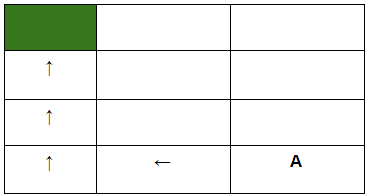
**The Bellman Equation**

Consider the following square of rooms which is analogous to the actual environment from our original problem but without the barriers.



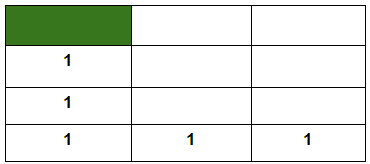
**An empty environment**

Now suppose a robot needs to go to the room, marked in green from its current position (A) using the specified direction.



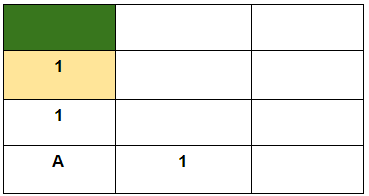
**Sample environment, agent and directions to proceed**

How can we enable the robot to do this programmatically? One idea would be to introduce some kind of footprint which the robot would be able to follow like below.



**An environment with value footprints**

Here, a constant value is specified in each of the rooms which will come along the robot’s way if it follows the direction specified above. In this way, if it starts at location A, it will be able to scan through this constant value and will move accordingly. But this would only work if the direction is prefixed and the robot always starts at location A.  
  
**Now, consider the robot starts at the following location:**



**An environment with value footprint (contd.)**

The robot now sees footprints in two different directions. It is, therefore, unable to decide which way to go in order to get to the destination (green room). It happens primarily because the robot does not have a way to remember the directions to proceed. So, our job now is to enable the robot with a **memory**.   
  
**This is where the Bellman Equation comes into play:**

V(s)=maxa(R(s,a)+γV(s′))V(s)=maxa(R(s,a)+γV(s′))

**Where:**

s = a particular state (room)

a = action (moving between the rooms)

s′ = state to which the robot goes from s

𝜸 = discount factor (we will get to it in a moment)

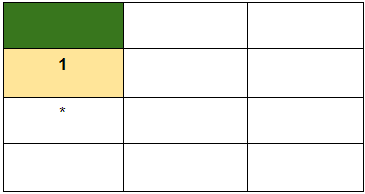
R(s, a) = a reward function which takes a state s and action a and outputs a reward value

V(s) = value of being in a particular state (the footprint)

We consider all the possible actions and take the one that yields the maximum value.

There is one constraint however regarding the value footprint i.e., the room, marked in yellow just below the green room will always have a value of 1 to denote that it is one of the nearest room adjacent to the green room. This is also to ensure that a robot gets a reward when it goes from the yellow room to the green room. Let’s now see how to make sense of the above equation here.

We will assume a discount factor of **0.9**.

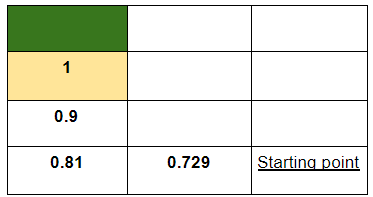


**An environment with value footprint (contd.)**

**\***For this room (read state) what will be V(s)? Let’s put the values into the equation straightly:

V(s)=maxa(0+0.9∗1)=0.9V(s)=maxa(0+0.9∗1)=0.9

Here, the robot will not get any reward for going to the state (room) marked in yellow, hence **R(s, a) = 0** here. The robot knows the values of being in the yellow room hence **V(s′)** is 1. Following this for the other states, we should get:



**An environment with some value footprints computed from the Bellman equation**

A couple of things to notice here:

* The **max()** function helps the robot to always choose the state that gives it the maximum value of being in that state.
* The **discount factor** 𝜸 notifies the robot about how far it is from the destination. This typically specified by the developer of the algorithm that would be instilled in the robot.
* The other states can also be given their respective values in a similar way:



**An environment with all the value footprints computed from the Bellman equation**

The robot now can proceed its way through the green room utilizing these value footprints even if it is dropped at any arbitrary room in the above square. Now, if a robot lands up in the highlighted (sky blue) room, it will still find two options to choose from. But eventually, either of the paths will be good enough for the robot to take because of the way the value footprints are now laid out.

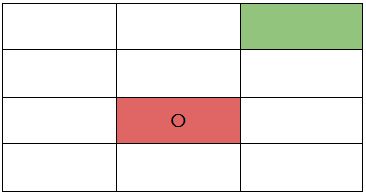
Notice how the main task of reaching a destination from a particular source got broken down to similar subtasks. It turns out that there is a particular programming paradigm developed especially for solving problems that have repetitive subproblems in them. It is referred to as [dynamic programming](about:blank). It was invented by [Richard Bellman](about:blank) in 1954 who also coined the equation we just studied (hence the name, Bellman Equation).

**Note**that this is one of the key equations in the world of reinforcement learning.

If we think realistically, our surroundings do not always work in the way we expect. There is always a bit of **stochasticity** involved in it. This applies to a robot as well. Sometimes, it might so happen that the robot’s inner machinery got corrupted. Sometimes, the robot may come across some hindrances on its way which may not be known to it beforehand. Sometimes, even if the robot knows that it needs to take the right turn, it will not. So, how do we introduce this stochasticity in our case? **Markov Decision Processes.**

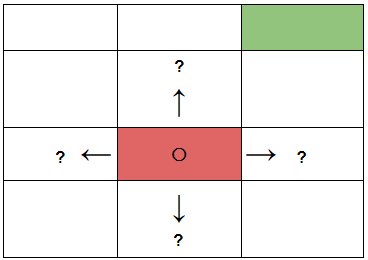
**Modeling stochasticity: Markov Decision Processes**

Consider the robot is currently in the red room and it needs to go to the green room.



**An environment with an agent**

Let’s now consider, the robot has a slight chance of dysfunctioning and might take the left or right or bottom turn instead of taking the upper turn in order to get to the green room from where it is now (red room). Now, the question is how do we enable the robot to handle this when it is out there in the above environment?



**An environment with an agent (with stochasticity)**

This is a situation where the decision making regarding which turn is to be taken is **partly random** and **partly under the control of the robot**. Partly random because we are not sure when exactly the robot might dysfunction and partly under the control of the robot because it is still making a decision of taking a turn on its own and with the help of the program embedded into it.

Here is the definition of **Markov Decision Processes** (collected from Wikipedia):

**A Markov decision process (MDP) is a discrete time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker.**

You may focus only on the highlighted part. We have the exact same situation here in our case.

We have now introduced ourselves with the concept of **partly random** and **partly controlled decision making**. We need to give this concept a mathematical shape (most likely an equation) which then can be taken further.

You might be surprised to see that we can do this with the help of the Bellman Equation with a few minor tweaks.

Here is the **original Bellman Equation**, again:

V(s)=maxa(R(s,a)+γV(s′))V(s)=maxa(R(s,a)+γV(s′))

What needs to be changed in the above equation so that we can introduce some amount of randomness here? As long as we are not sure when the robot might not take the expected turn, we are then also not sure in which room it might end up in which is nothing but the room it moves from its current room. At this point, according to the above equation, we are not sure of s′ which is the next state (room, as we were referring them).

But we do know all the probable turns the robot might take! In order to incorporate each of these probabilities into the above equation, **we need to associate a probability with each of the turns to quantify that the robot has got x% chance of taking this turn**.   
  
If we do so, we get:

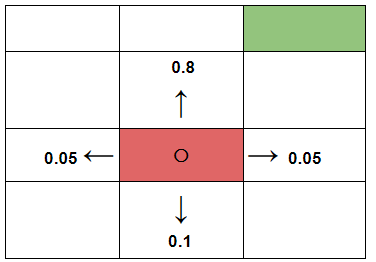
V(s)=maxa(R(s,a)+γ∑s′P(s,a,s′)V(s′))V(s)=maxa(R(s,a)+γ∑s′P(s,a,s′)V(s′))

**Taking the new notations step by step:**

**P(s, a, s′)**- the probability of moving from room **s** to room **s′** with action **a**

∑s′P(s,a,s′)V(s′)∑s′P(s,a,s′)V(s′)- [expectation](about:blank)of the situation that the robot incurs randomness.

Notice, **everything else from the above two points is exactly the same**. Let’s assume the following probabilities are associated with each of the turns the robot might take while being in that red room (to go to the green room).



**An environment with an agent (with probabilities)**

When we associate probabilities to each of these turns, **we essentially mean that there is an 80% chance that the robot will take the upper turn**. If we put all the required values in our equation, we get:

V(s)=maxa(R(s,a)+γ((0.8V(roomup))+(0.1V(roomdown))+...))V(s)=maxa(R(s,a)+γ((0.8V(roomup))+(0.1V(roomdown))+...))

**Note**that the value footprints will now change due to the fact that we are incorporating stochasticity here. But this time, we will not calculate those value footprints. Instead, we will let the robot to figure it out (more on this in a moment).

Up until this point, we have not considered about rewarding the robot for its action of going into a particular room. We are only rewarding the robot when it gets to the destination. **Ideally, there should be a reward for every action the robot takes to help it better assess the quality of its actions**. The rewards need not be always the same. But it is much better than having some amount reward for the actions than having no rewards at all. This idea is known as the **living penalty**. In reality, the rewarding system can be very complex and particularly [modeling sparse rewards](https://www.youtube.com/watch?v=0Ey02HT_1Ho) is an active area of research in the domain reinforcement learning.   
  
If you would like to give a spin to this topic then following resources might come in handy:

[Curiosity-driven Exploration by Self-supervised Prediction](https://pathak22.github.io/noreward-rl/)

[Curiosity and Procrastination in Reinforcement Learning](file:///F:\UMGC\Data640%20-%20Spring%202021\Lessons%20-%20revised\Lesson%209%20-%20Intro%20to%20RL\Reinforcement%20Learning%20docs%20from%20Ed\debug.log)

[Learning to Generalize from Sparse and Underspecified Rewards](https://ai.googleblog.com/2019/02/learning-to-generalize-from-sparse-and.html)

In the next section, we will introduce the notion of the **quality** of an action rather than looking at the value of going into a particular room (**V(s)**).

**Transitioning to Q-Learning**

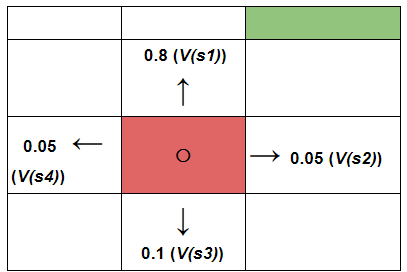
By now, we have got the following equation which gives us a value of going to a particular state (form now on, we will refer to the rooms as states) taking the stochasticity of the environment into the account:

V(s)=maxa(R(s,a)+γ∑s′P(s,a,s′)V(s′))V(s)=maxa(R(s,a)+γ∑s′P(s,a,s′)V(s′))

We have also learned very briefly about the idea of **living penalty** which deals with associating each move of the robot with a reward.

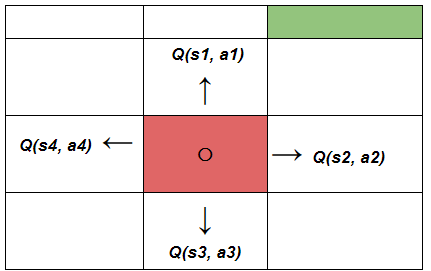
Q-Learning poses an idea of assessing the **quality of an action** that is taken to move to a state rather than determining the possible value of the state (value footprint) being moved to.

**Earlier, we had:**



**An environment with an agent (with possible value footprints)**

If we incorporate the idea of assessing the quality of actions for moving to a certain state**s′**.



**An environment with an agent (with quality of actions)**

The robot now has four different states to choose from and along with that, there are four different actions also for the current state it is in. So how do we calculate **Q(s, a)**i.e. the cumulative quality of the possible actions the robot might take?

From this equation:

V(s)=maxa(R(s,a)+γ∑s′P(s,a,s′)V(s′))V(s)=maxa(R(s,a)+γ∑s′P(s,a,s′)V(s′)), **if we discard the max() function, we get:**

R(s,a)+γ∑s′(P(s,a,s′)V(s′))R(s,a)+γ∑s′(P(s,a,s′)V(s′))

Essentially, in the equation that produces **V(s)**, we are considering all possible actions and all possible states (from the current state the robot is in) and then we are taking the maximum value caused by taking a certain action. The above equation produces a value footprint is for just one possible action. In fact, we can think of it as the quality of the action:

Q(s,a)=R(s,a)+γ∑s′(P(s,a,s′)V(s′))Q(s,a)=R(s,a)+γ∑s′(P(s,a,s′)V(s′))

Now that we have got an equation to quantify the quality of a particular action we are going to make a little adjustment in the above equation. We can now say that**V(s)** is the maximum of all the possible values of ***Q(s, a)***.   
  
Let’s utilize this fact and replace ***V(s′)*** as a function of ***Q()***:

Q(s,a)=R(s,a)+γ∑s′(P(s,a,s′)maxa′Q(s′,a′))Q(s,a)=R(s,a)+γ∑s′(P(s,a,s′)maxa′Q(s′,a′))

But why would we do that? **To ease our calculations**. Because now, we have only one function **Q()** (which is also at the core of the dynamic programming paradigm) to calculate and **R(s, a) is a quantified metric** which produces rewards of moving to a certain state.

The qualities of the actions are called **Q-values**. And from now on, we will refer the value footprints as the Q-values.

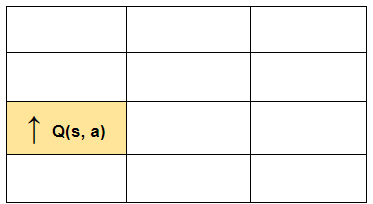
We now have the last piece of the puzzle remaining i.e., [**temporal difference**](https://medium.com/@violante.andre/simple-reinforcement-learning-temporal-difference-learning-e883ea0d65b0)before we jump to the implementation part and we are going to study that in the next section.

**The last piece of the puzzle: Temporal difference**

Recollect the statement from a previous section:

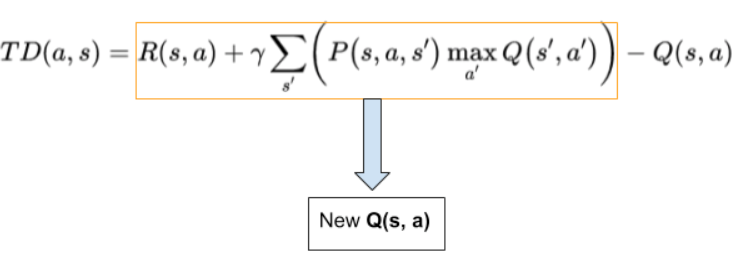
But this time, we will not calculate those value footprints. Instead, we will let the robot to figure it out (more on this in a moment).

Temporal Difference is the component that will help the robot to calculate the Q-values with respect to the changes in the environment over time. Consider our robot is currently in the marked state and it wants to move to the upper state. Note that the robot already knows the Q-value of making the action i.e., moving to the upper state.



**An environment with an agent**

We know that the environment is stochastic in nature and the reward that the robot will get after moving to the upper state might be different from an earlier observation. So how do we capture this change (read difference)? We recalculate the new **Q(s, a)**with the same formula and subtract the previously known **Q(s, a)**from it.



The equation that we just derived gives the temporal difference in the Q-values which further helps to capture the random changes that the environment may impose. The new **Q(s, a)** is updated as the following:

Qt(s,a)=Qt−1(s,a)+αTDt(a,s)Qt(s,a)=Qt−1(s,a)+αTDt(a,s)

where,

ɑ is the learning rate which controls how quickly the robot adopts to the random changes imposed by the environment

Qt(s,a)Qt(s,a)is the current Q-value

Qt−1(s,a)Qt−1(s,a)is the previously recorded Q-value

If we replaceTDt(s,a)TDt(s,a)with its full-form equation, we should get:

Qt(s,a)=Qt−1(s,a)+α(R(s,a)+γmaxa′Q(s′,a′)−Qt−1(s,a))Qt(s,a)=Qt−1(s,a)+α(R(s,a)+γmaxa′Q(s′,a′)−Qt−1(s,a))

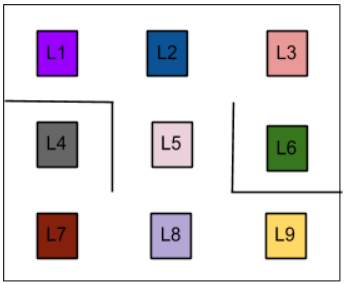
**We now have all the little pieces of Q-Learning together to move forward to its implementation part.**

Feel free to review the problem statement once which we discussed in the very beginning.

**Implementing Q-Learning in Python with Numpy**

If you do not have a local setup, you can run [this notebook](about:blank) directly on FloydHub by just clicking on the below button -

To implement the algorithm, we need to understand the warehouse locations and how that can be mapped to different states. Let’s start by recollecting the sample environment shown earlier:



**A sample environment**

Let’s map each of the above locations in the warehouse to numbers (states). It will ease our calculations.

# Define the states

location\_to\_state = {

'L1' : 0,

'L2' : 1,

'L3' : 2,

'L4' : 3,

'L5' : 4,

'L6' : 5,

'L7' : 6,

'L8' : 7,

'L9' : 8

}

We will only be using Numpy as our dependency. So let’s import that aliased as **np:**

import numpy as np

The next step is to define the actions which as mentioned above represents the transition to the next state:

# Define the actions

actions = [0,1,2,3,4,5,6,7,8]

Now the reward table:

# Define the rewards

rewards = np.array([[0,1,0,0,0,0,0,0,0],

[1,0,1,0,1,0,0,0,0],

[0,1,0,0,0,1,0,0,0],

[0,0,0,0,0,0,1,0,0],

[0,1,0,0,0,0,0,1,0],

[0,0,1,0,0,0,0,0,0],

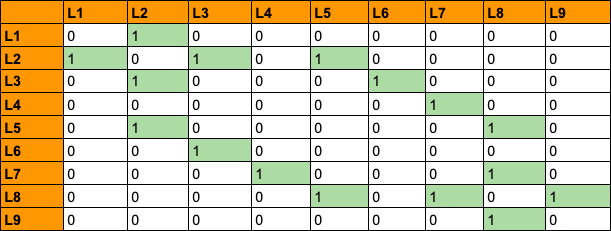
[0,0,0,1,0,0,0,1,0],

[0,0,0,0,1,0,1,0,1],

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

If you understood it correctly, there isn't any real barrier limitation as depicted in the image. For example, the transition L4 to L1 is allowed but the reward will be zero to discourage that path.

In the above code snippet, we took each of the states and put ones in the respective state that are directly reachable from a certain state. Refer to the reward table once again. The above array construction will be easy to understand then. Note that, we did not consider the top-priority location (**L6**) yet.

**Table of rewards (rewards matrix)**

We would also need an inverse mapping from the states back to original location indicators. It will be clearer when we reach to the utter depths of the algorithm.

The following line of code will do this for us.

# Maps indices to locations

state\_to\_location = dict((state,location) for location,state in location\_to\_state.items())

Before we write a helper function which will yield the optimal route for going from one location to another, let’s specify the two parameters of the Q-Learning algorithm - ɑ (learning rate) and 𝜸 (discount factor).

# Initialize parameters

gamma = 0.75 # Discount factor

alpha = 0.9 # Learning rate

We will now define a function **get\_optimal\_route()** which will:

Take two arguments:

starting location in the warehouse and

end location in the warehouse respectively

And will return the optimal route for reaching the end location from the starting location in the form of an ordered list containing the letters

We will start defining the function by initializing the Q-values to be all zeros.

# Initializing Q-Values

Q = np.array(np.zeros([9,9]))

For convenience, we will copy the rewards matrix **rewards**to a separate variable and will operate on that.

# Copy the rewards matrix to new Matrix

rewards\_copy = np.copy(rewards)

[9,9] as we have a total of 9 locations. The function will take a starting location and an ending location. So, let’s set the priority of the ending location to a larger integer like 999 and get the ending state from the location letter (such as **L1**, **L2**and so on).

# Get the ending state corresponding to the ending location as given

ending\_state = location\_to\_state[end\_location]

# With the above information automatically set the priority of the

# given ending state to the highest one

rewards\_copy[ending\_state,ending\_state] = 999

Learning is a continuous process, hence we will let the robot to explore the environment for a while and we will do it by simply looping it through 1000 times. We will then pick a state randomly from the set of states we defined above and we will call it **current\_state**.

for i in range(1000):

# Pick up a state randomly

current\_state = np.random.randint(0,9)

We then, being inside the loop, iterate through the rewards matrix to get the states that are directly reachable from the randomly chosen current state and we will assign those state in a list named **playable\_actions**.

**Note** that so far we have not bothered about the starting location yet.

playable\_actions = []

# Iterate through the new rewards matrix and get the actions > 0

for j in range(9):

if rewards\_copy[current\_state,j] > 0:

playable\_actions.append(j)

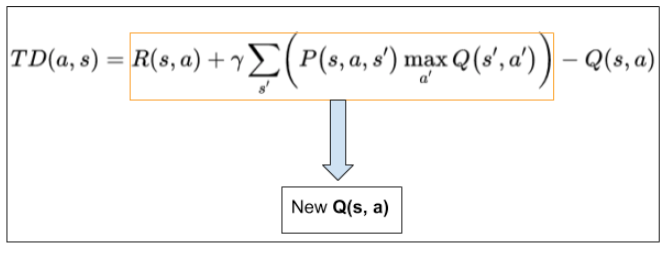
We will now choose a state randomly from the **playable\_actions**.

# Pick an action randomly from the list of playable actions leading us to the next

# state

next\_state = np.random.choice(playable\_actions)

We can now compute the temporal difference and update the Q-values accordingly. Here is the formula of temporal difference for your convenience:



Here is the way to update the Q-values using the temporal difference:

Qt(s,a)=Qt−1(s,a)+αTDt(a,s)Qt(s,a)=Qt−1(s,a)+αTDt(a,s)

# Compute the temporal difference

# The action here exactly refers to going to the next state

TD = rewards\_copy[current\_state,next\_state] + gamma \* Q[next\_state, np.argmax(Q[next\_state,])] - Q[current\_state,next\_state]

# Update the Q-Value using the Bellman equation

Q[current\_state,next\_state] += alpha \* TD

The good news is we are done implementing the most critical part of the process and up until now the definition of **get\_optimal\_route()**should look like:

def get\_optimal\_route(start\_location,end\_location):

# Copy the rewards matrix to new Matrix

rewards\_new = np.copy(rewards)

# Get the ending state corresponding to the ending location as given

ending\_state = location\_to\_state[end\_location]

# With the above information automatically set the priority of the given ending

# state to the highest one

rewards\_new[ending\_state,ending\_state] = 999

# -----------Q-Learning algorithm-----------

# Initializing Q-Values

Q = np.array(np.zeros([9,9]))

# Q-Learning process

for i in range(1000):

# Pick up a state randomly

current\_state = np.random.randint(0,9) # Python excludes the upper bound

# For traversing through the neighbor locations in the maze

playable\_actions = []

# Iterate through the new rewards matrix and get the actions > 0

for j in range(9):

if rewards\_new[current\_state,j] > 0:

playable\_actions.append(j)

# Pick an action randomly from the list of playable actions

# leading us to the next state

next\_state = np.random.choice(playable\_actions)

# Compute the temporal difference

# The action here exactly refers to going to the next state

TD = rewards\_new[current\_state,next\_state] + gamma \* Q[next\_state, np.argmax(Q[next\_state,])] - Q[current\_state,next\_state]

# Update the Q-Value using the Bellman equation

Q[current\_state,next\_state] += alpha \* TD

We will now start the other half of finding the optimal route. We will first initialize the optimal route with the starting location.

# Initialize the optimal route with the starting location

route = [start\_location]

We currently do not know about the next move of the robot. Thus we will set the next location to also be the starting location.

next\_location = start\_location

Since we do not know the exact number of iterations the robot will take in order to find out the optimal route, we will simply loop the next set of processes until the next location is not equal to the ending location. This is where we will terminate the loop.

# We don't know about the exact number of iterations needed to reach to the final

# location hence while loop will be a good choice for iteratiing

while(next\_location != end\_location):

# Fetch the starting state

starting\_state = location\_to\_state[start\_location]

# Fetch the highest Q-value pertaining to starting state

next\_state = np.argmax(Q[starting\_state,])

# We got the index of the next state. But we need the corresponding letter.

next\_location = state\_to\_location[next\_state]

route.append(next\_location)

# Update the starting location for the next iteration

start\_location = next\_location

return route

Finally, we return the route.

return route

Here is the whole**get\_optimal\_route()**function:

def get\_optimal\_route(start\_location,end\_location):

# Copy the rewards matrix to new Matrix

rewards\_new = np.copy(rewards)

# Get the ending state corresponding to the ending location as given

ending\_state = location\_to\_state[end\_location]

# With the above information automatically set the priority of

# the given ending state to the highest one

rewards\_new[ending\_state,ending\_state] = 999

# -----------Q-Learning algorithm-----------

# Initializing Q-Values

Q = np.array(np.zeros([9,9]))

# Q-Learning process

for i in range(1000):

# Pick up a state randomly

current\_state = np.random.randint(0,9) # Python excludes the upper bound

# For traversing through the neighbor locations in the maze

playable\_actions = []

# Iterate through the new rewards matrix and get the actions > 0

for j in range(9):

if rewards\_new[current\_state,j] > 0:

playable\_actions.append(j)

# Pick an action randomly from the list of playable actions

# leading us to the next state

next\_state = np.random.choice(playable\_actions)

# Compute the temporal difference

# The action here exactly refers to going to the next state

TD = rewards\_new[current\_state,next\_state] + gamma \* Q[next\_state, np.argmax(Q[next\_state,])] - Q[current\_state,next\_state]

# Update the Q-Value using the Bellman equation

Q[current\_state,next\_state] += alpha \* TD

# Initialize the optimal route with the starting location

route = [start\_location]

# We do not know about the next location yet, so initialize with the value of

# starting location

next\_location = start\_location

# We don't know about the exact number of iterations

# needed to reach to the final location hence while loop will be a good choice

# for iteratiing

while(next\_location != end\_location):

# Fetch the starting state

starting\_state = location\_to\_state[start\_location]

# Fetch the highest Q-value pertaining to starting state

next\_state = np.argmax(Q[starting\_state,])

# We got the index of the next state. But we need the corresponding letter.

next\_location = state\_to\_location[next\_state]

route.append(next\_location)

# Update the starting location for the next iteration

start\_location = next\_location

return route

If we call **print(get\_optimal\_route('L9', 'L1'))**, we should get:

['L9', 'L8', 'L5', 'L2', 'L1']

Note how the program considers the barriers that are present in the environment. For fun, you can change the ɑ and 𝜸 parameters to see how the learning process changes. With this, let me move to the conclusion section where we will be discussing how you can take your reinforcement learning journey further.

Let’s refactor the code a little bit to conform to **OOP**paradigm. We will define a class named **QAgent()**containing the following two methods apart from init:

**training(self, start\_location, end\_location, iterations)** which will help the robot to obtain the Q-values from the environment

**get\_optimal\_route(self, start\_location, end\_location, next\_location, route, Q)** which will get the robot an optimal route from a point to another

Let’s first define the **\_\_init\_\_()** method which would initialize the class constructor:

def \_\_init\_\_(self, alpha, gamma, location\_to\_state, actions, rewards, state\_to\_location, Q):

self.gamma = gamma

self.alpha = alpha

self.location\_to\_state = location\_to\_state

self.actions = actions

self.rewards = rewards

self.state\_to\_location = state\_to\_location

self.Q = Q

Now comes the **training()**method:

def training(self, start\_location, end\_location, iterations):

rewards\_new = np.copy(self.rewards)

ending\_state = self.location\_to\_state[end\_location]

rewards\_new[ending\_state, ending\_state] = 999

for i in range(iterations):

current\_state = np.random.randint(0,9)

playable\_actions = []

for j in range(9):

if rewards\_new[current\_state,j] > 0:

playable\_actions.append(j)

next\_state = np.random.choice(playable\_actions)

TD = rewards\_new[current\_state,next\_state] +

self.gamma \* self.Q[next\_state, np.argmax(self.Q[next\_state,])] - self.Q[current\_state,next\_state]

self.Q[current\_state,next\_state] += self.alpha \* TD

route = [start\_location]

next\_location = start\_location

# Get the route

self.get\_optimal\_route(start\_location, end\_location, next\_location, route, self.Q)

Finally the **get\_optimal\_route()** method.

def get\_optimal\_route(self, start\_location, end\_location, next\_location, route, Q):

while(next\_location != end\_location):

starting\_state = self.location\_to\_state[start\_location]

next\_state = np.argmax(Q[starting\_state,])

next\_location = self.state\_to\_location[next\_state]

route.append(next\_location)

start\_location = next\_location

print(route)

The entire class definition should look like:

class QAgent():

# Initialize alpha, gamma, states, actions, rewards, and Q-values

def \_\_init\_\_(self, alpha, gamma, location\_to\_state, actions, rewards, state\_to\_location, Q):

self.gamma = gamma

self.alpha = alpha

self.location\_to\_state = location\_to\_state

self.actions = actions

self.rewards = rewards

self.state\_to\_location = state\_to\_location

self.Q = Q

# Training the robot in the environment

def training(self, start\_location, end\_location, iterations):

rewards\_new = np.copy(self.rewards)

ending\_state = self.location\_to\_state[end\_location]

rewards\_new[ending\_state, ending\_state] = 999

for i in range(iterations):

current\_state = np.random.randint(0,9)

playable\_actions = []

for j in range(9):

if rewards\_new[current\_state,j] > 0:

playable\_actions.append(j)

next\_state = np.random.choice(playable\_actions)

TD = rewards\_new[current\_state,next\_state] + \

self.gamma \* self.Q[next\_state, np.argmax(self.Q[next\_state,])] - self.Q[current\_state,next\_state]

self.Q[current\_state,next\_state] += self.alpha \* TD

route = [start\_location]

next\_location = start\_location

# Get the route

self.get\_optimal\_route(start\_location, end\_location, next\_location, route, self.Q)

# Get the optimal route

def get\_optimal\_route(self, start\_location, end\_location, next\_location, route, Q):

while(next\_location != end\_location):

starting\_state = self.location\_to\_state[start\_location]

next\_state = np.argmax(Q[starting\_state,])

next\_location = self.state\_to\_location[next\_state]

route.append(next\_location)

start\_location = next\_location

print(route)

Once the class is compiled, you should be able to create a class object and call the **training()**method like so:

qagent = QAgent(alpha, gamma, location\_to\_state, actions, rewards, state\_to\_location, Q)

qagent.training('L9', 'L1', 1000)

Notice that every is exactly similar to previous chunk of code but the refactored version indeed looks more elegant and modular. And yes, the output will also be same.

['L9', 'L8', 'L5', 'L2', 'L1']

**Conclusion and further steps**

We finally have come to the very end of the article. We covered a lot of preliminary grounds of reinforcement learning that will be useful if you are planning to further strengthen your knowledge of reinforcement learning. We also implemented the simplest reinforcement learning just by using **Numpy**. These base scratch implementations are not only for just fun but also they help tremendously to know the nuts and bolts of an algorithm.

Reinforcement learning has given solutions to many problems from a wide variety of different domains. One that I particularly like is [Google’s **NasNet**](about:blank) which uses deep reinforcement learning for finding an optimal neural network architecture for a given dataset.  

Let’s now review some of the best resources for breaking into reinforcement learning in a serious manner:

[Reinforcement Learning, Second Edition: An Introduction](about:blank#reinforcement-learning-intro) by **Richard S. Sutton** and **Andrew G. Barto**which is considered to be the textbook of reinforcement learning

[Practical Reinforcement Learning](about:blank) a course designed by the National Research University Higher School of Economics offered by Coursera

[Reinforcement Learning](about:blank) a course designed by the Georgia  University and offered by Udacity

If you are interested in the conjunction of meta-learning and reinforcement learning then you may follow [this article](about:blank)

How about combining deep learning + reinforcement learning? Check out [Deep RL Bootcamp](about:blank).

[Deep Reinforcement Learning Hands-On](about:blank#deep-reinforcement-learning-hands-on) a book by Maxim Lapan which covers many cutting edge RL concepts like deep Q-networks, value iteration, policy gradients and so on.

[MIT Deep Learning](about:blank#intro-to-rl) a course taught by Lex Fridman which teaches you how different deep learning applications are used in autonomous vehicle systems and more

[Introduction to Reinforcement Learning](about:blank#adv-dl-rl) a course taught by one of the main leaders in the game of reinforcement learning - David Silver

[Spinning Up in Deep RL](about:blank#spinning-up) a course offered from the house of OpenAI which serves  as your guide to connecting the dots between theory and practice in deep reinforcement learning

[Controlling a 2D Robotic Arm with Deep Reinforcement Learning](about:blank) an article which shows how to build your own robotic arm best friend by diving into deep reinforcement learning

[Spinning Up a Pong AI With Deep Reinforcement Learning](about:blank) an article which shows you to code a vanilla policy gradient model that plays the beloved early 1970s classic video game Pong in a step-by-step manner

The list is kind of handpicked for those who really want to step up their game in reinforcement learning. So if you are one among them, don’t forget to check out the resources.

I hope you enjoyed the article and you will take it forward to make applications that can adapt with respect to the environment they are employed to.

Thanks to [Alessio](about:blank) and [Bharath](about:blank) of FloydHub for sharing their valuable feedback on the article. Big thanks to the entire FloydHub team for letting me run the accompanying notebook on their platform. If you haven’t checked FloydHub yet, give FloydHub a spin for your Machine Learning and Deep Learning projects. There is no going back once you’ve learned how easy they make it.

**FloydHub Call for AI writers**

Want to write amazing articles like Sayak and play your role in the long road to Artificial General Intelligence? [We are looking for passionate writers](about:blank), to build the world's best blog for practical applications of groundbreaking A.I. techniques. FloydHub has a large reach within the AI community and with your help, we can inspire the next wave of AI. [Apply now](about:blank) and join the crew!

**About Sayak Paul**  
Sayak loves everything deep learning. He goes by the motto of understanding complex things and helping people understand them as easily as possible. Sayak is an extensive blogger and all of his blogs can be found [here](about:blank). He is also working with his friends on the application of deep learning in Phonocardiogram classification. Sayak is also a FloydHub AI Writer. He is always open to discussing novel ideas and taking them forward to implementations. You can connect with Sayak on [LinkedIn](about:blank) and [Twitter](about:blank).

# Appendix2: Q-Learning using numpy Code by Sayak Paul

#source: [https://github.com/sayakpaul/FloydHub-Q-Learning-Blog/blob/master/Q-Learning%20using%20numpy.ipynb](about:blank)

# Only numpy

import numpy as np

# Initialize parameters

gamma = 0.9 # Discount factor

alpha = 0.75 # Learning rate

# Define the states

location\_to\_state = {

    'L1' : 0,

    'L2' : 1,

    'L3' : 2,

    'L4' : 3,

    'L5' : 4,

    'L6' : 5,

    'L7' : 6,

    'L8' : 7,

    'L9' : 8

}

# Define the actions

actions = [0,1,2,3,4,5,6,7,8]

# Define the rewards

rewards = np.array([[0,1,0,0,0,0,0,0,0],

              [1,0,1,0,0,0,0,0,0],

              [0,1,0,0,0,1,0,0,0],

              [0,0,0,0,0,0,1,0,0],

              [0,1,0,0,0,0,0,1,0],

              [0,0,1,0,0,0,0,0,0],

              [0,0,0,1,0,0,0,1,0],

              [0,0,0,0,1,0,1,0,1],

              [0,0,0,0,0,0,0,1,0]])

# Maps indices to locations

state\_to\_location = dict((state,location) for location,state in location\_to\_state.items())

def get\_optimal\_route(start\_location,end\_location):

    # Copy the rewards matrix to new Matrix

    rewards\_new = np.copy(rewards)

    # Get the ending state corresponding to the ending location as given

    ending\_state = location\_to\_state[end\_location]

    # With the above information automatically set the priority of the given ending state to the highest one

    rewards\_new[ending\_state,ending\_state] = 999

    # -----------Q-Learning algorithm-----------

    # Initializing Q-Values

    Q = np.array(np.zeros([9,9]))

    # Q-Learning process

    for i in range(1000):

        # Pick up a state randomly

        current\_state = np.random.randint(0,9) # Python excludes the upper bound

        # For traversing through the neighbor locations in the maze

        playable\_actions = []

        # Iterate through the new rewards matrix and get the actions > 0

        for j in range(9):

            if rewards\_new[current\_state,j] > 0:

                playable\_actions.append(j)

        # Pick an action randomly from the list of playable actions leading us to the next state

        next\_state = np.random.choice(playable\_actions)

        # Compute the temporal difference

        # The action here exactly refers to going to the next state

        TD = rewards\_new[current\_state,next\_state] + gamma \* Q[next\_state, np.argmax(Q[next\_state,])] - Q[current\_state,next\_state]

        # Update the Q-Value using the Bellman equation

        Q[current\_state,next\_state] += alpha \* TD

    # Initialize the optimal route with the starting location

    route = [start\_location]

    # We do not know about the next location yet, so initialize with the value of starting location

    next\_location = start\_location

    # We don't know about the exact number of iterations needed to reach to the final location hence while loop will be a good choice for iteratiing

    while(next\_location != end\_location):

        # Fetch the starting state

        starting\_state = location\_to\_state[start\_location]

        # Fetch the highest Q-value pertaining to starting state

        next\_state = np.argmax(Q[starting\_state,])

        # We got the index of the next state. But we need the corresponding letter.

        next\_location = state\_to\_location[next\_state]

        route.append(next\_location)

        # Update the starting location for the next iteration

        start\_location = next\_location

    return route

print(get\_optimal\_route('L9', 'L1'))