Adrien Kaczmarek
Hiroshi Esaki
12 July 2020

Reinforcement Learning, adaptivity to unknown situations

Concepts and Applications

Nowadays, Machine Learning has become a famous field in Computer Science. It succeeds in numerous domains such as Image Recognition or Data Forecast. One branch of Machine Learning is still less known than the others: Reinforcement Learning. In the next few years, this branch of Machine Learning could bring new solutions to the current problem.

I will explain in outline what is reinforcement learning, and create an algorithm from scratch. Then I will play with this algorithm to demonstrate how well it is adaptive to unseen situations, and thus what is Reinforcement Learning real potential.

What is Reinforcement Learning?

Presentation of Machine Learning

Machine Learning is usually divided into three branches.

Unsupervised Learning: Algorithms that find similitude between objects (e.g. numbers, people's data).

Supervised Learning: Algorithms that converge to a function that maps two known spaces (e.g. space of houses information to a space of prices). Among all the algorithms, the hype is nowadays around Neural Network's algorithms.

Reinforcement Learning: Algorithms that converge to a function that a not necessarily known space to a known space (e.g. a space of sequences of game states to a space of actions to perform). These algorithms could be very straight forward when both spaces are entirely known. These algorithms could use other parts of Machine Learning into them such as the Deep Neural Network.

Here "known" is used to mean that the dimension of the space is known and the space size is sufficiently small.

More commonly, Unsupervised Learning is used to create a cluster in given data, Supervised Learning is used to evaluate the value of given data and Reinforcement Learning is used to perform actions in a given environment.

Reinforcement Learning, Agent and Environment

In Reinforcement Learning, we trained an Agent to perform valuable actions in a given Environment.

Thus, for each action the Agent performs, the Environment sends back a reward. This reward could be negative, null, or positive. The more the value of the reward is high the more the agent is in a valuable state.

Example of Reinforcement Learning Algorithm

Reinforcement Learning is quite a new technology. However, it begins to be used in a lot of different projects.

The most famous of them is AlphaGo, that beat Lee Sedol in 2016 (SILVER, DAVID, HUANG, AJA, MADDISON, CHRIS J., ET AL. MASTERING THE GAME OF GO WITH DEEP NEURAL NETWORKS AND TREE SEARCH. NATURE, 2016, VOL. 529, NO 7587, P. 484-489.).

These algorithms are also used to bet video games, like ATARI games (MNIH, VOLODYMYR, KAVUKCUOGLU, KORAY, SILVER, DAVID, ET AL. PLAYING ATARI WITH DEEP REINFORCEMENT LEARNING. ARXIV PREPRINT ARXIV:1312.5602, 2013.).

Recently, the university of Berkeley used Reinforcement Learning to learn a robot to walk on a wide variety of surface (PENG, XUE BIN, COUMANS, ERWIN, ZHANG, TINGNAN, ET AL. LEARNING AGILE ROBOTIC LOCOMOTION SKILLS BY IMITATING ANIMALS. ARXIV PREPRINT ARXIV:2004.00784, 2020.).

A few Web Applied Examples

Web Spiders

(RENNIE, JASON, MCCALLUM, ANDREW, ET AL. USING REINFORCEMENT LEARNING TO SPIDER THE WEB EFFICIENTLY. IN : ICML. 1999. P. 335-343.)

A Spider is used to explore the Web by following hyperlinks. This is then used to construct Web Browsers. However, the Spider has to choose the good hyperlinks. Here the Spider is the Agent and the Web is the Environment. The actions to perform are the choice of the hyperlinks to follow.

However, these technologies are not widely used nowadays for creating common Web Crawlers.

Web Advertising

(CAI, HAN, REN, KAN, ZHANG, WEINAN, ET AL. REAL-TIME BIDDING BY REINFORCEMENT LEARNING IN DISPLAY ADVERTISING. IN : PROCEEDINGS OF THE TENTH ACM INTERNATIONAL CONFERENCE ON WEB SEARCH AND DATA MINING. 2017. P. 661-670.)

To deliver the best ads possible, the company needs to evaluate how much money the user can spend and estimate how much money the ads campaign will cost the company.Here the Agent chooses when to display ads to the user and what ads to display and the Environment is the Web Market.

These technologies are widely used on the Internet.

Internet Congestion Control

(JAY, NATHAN, ROTMAN, NOGA, GODFREY, BRIGHTEN, ET AL. A DEEP REINFORCEMENT LEARNING PERSPECTIVE ON INTERNET CONGESTION CONTROL. IN : INTERNATIONAL CONFERENCE ON MACHINE LEARNING. 2019. P. 3050-3059.)

To distribute the information inside a network, sender and receiver have to adjust their transmission rate to the network velocity. The more information passed through the network the more the transmission rate has to be low, if the transmission rate is too high, there could be data leak. The main problem here is to predict when the network will be congested to adjust right in time the transmission rate. The congestion control protocol used nowadays is not highly efficient. In this paper, they tried a new congestion control protocol based on Reinforcement Learning.

Thus, the Agent chooses the transmission rate in the network Environment to maximise the transmission rate with at least leak as possible.

These are new technologies, not yet commonly used through the Internet.

Basic Code Project

In this section, we will build a simple algorithm using Reinforcement Learning. This algorithm has to play a game named Frozen Lake.

Rules of the game

The rules are the following.

The game is played on a 4x4 grid that represents the world.

At the beginning of the game, the player starts at the top left corner.

To win the game, the player has to reach the bottom right corner.

In the grid there are holes. If the player moves into a hole it loses.

The player can move in four directions: up, right, down, left.

When the player moves in a certain direction, it will move to the expected direction with probability 1/3 and move to one of the perpendicular directions with probability 2/3.

The last rule is the most important. Indeed, without this rule, a simple path-finding algorithm could play this game. However, the game is not deterministic, thus, it could be challenging even for a real human.

Screenshot of the game FrozenLake. The player is denoted by the letter S, the goal is denoted by the letter G, holes are denoted by the letter H and walkable areas are denoted by the letter F.

Here, the Agent will be the player and the Environment will be the 4x4 grid. Because we want the Agent to reach the goal, the Environment will send a reward positive to the Agent when it is in the goal position. Depending on which kind of training we are doing, we can send a negative reward to the Agent when it reaches a hole. We will detail these considerations later on.

Basics of Q-Learning

Before going deep into the code, we will briefly explain the main concepts behind Reinforcement Learning.

An Environment corresponds to a bunch of states on which the Agent can be. To each state we are going to give a Value, the more its value is high the more the state is valuable for the Agent to be. However, we do not know anything about these Values at the beginning of the Training.

For example, here the states are the position the player can reach, where (1, 1) is the initial position of the player and (4, 4) is the goal position.

Some algorithms deal with the Value of the states, however, they are not the easiest one. Instead, we can look at the Policy. To illustrate what the Policy is let us imagine the following scenes.

We are in state A, from there we can go either to state B by acting I or to state C by acting II. We want to know which action to perform among I and II. A Policy takes two arguments, the state in which we are and the action we perform in this state. If the Value of state B is greater than the Value of state C then we should act I. That means that Policy(A, I) > Policy(A, II). Otherwise, we should act II. That means that Policy(A, II) > Policy(A, I). We are now able to choose which action to perform without knowing any Values, only knowing the Policy is enough.

However the problem seems the same, we still have something we do not know. Moreover, the Policy has more unknown values than Values. Indeed, if we have N states and P actions, we have N Values and P x N Policies.

Though, it is way easier to find a mathematical formula for the Policy.

Commonly, the Policy is called and the reward sent by the Environment at time step is called.

In our problem, the number of states is discrete and finite (sixteen, one for each position). Thus, the Policy can be easily translated as a table of 16x4 elements - 16 for the numbers of states and 4 for the numbers of actions (four directions) -. This table is called the Q-Table.

Before going into the mathematical formula, there are two hyper-parameters we need to know. A hyper-parameter is a variable we have to choose by hand before the Training Phase.

The first one is the Learning rate called. It is a real between 0 and 1. The closer it is to 1, the more the algorithm will forget the previous iteration of its Training. If the algorithm does not forget at all its previous training phases, it will not learn at all. Whereas if the algorithm forgets all that it learned previously it will never improve.

The second one is the Discount Rate called. It is a real between 0 and 1. The closer it is to 1, the more the algorithm will look ahead for the next action to perform. If the algorithm does not look ahead at all it could move close to a hole and then increase its chance to lose. Whereas if the algorithm looks too far ahead, it will not find immediate danger.

Both these hyper-parameters influence a lot the Training. If they are not well chosen the algorithm will never succeed. Thus, many iterations are needed to find a good couple of values.

Bellman's Equation gave us a way to approximate the Policy through iterations. To understand where this equation comes from, there are numerous resources on the Internet like deeplizard.com (step by step path through Reinforcement Learning in Python3) among others.

$$q_*(s,a) = E\left[(1-\alpha)q_*(s,a) + \alpha\left(R_{t+1} + \gamma \max_{a'} q_*(s',a')\right)\right]$$

Bellman's Equation, s and a are shortcuts for state and action, s' is the state reached by the Agent after acting a.

To approximate the Policy, we will perform as many iterations as possible. To do so, we will play the game many times, each play is called an Episode.

The Bellman's Equation does not mind if there is multiple Episode, it just looks at the action performed at time t and the reward received for performing this action (the reward is sent by the Environment at time t and received by the Agent at time t + 1).

Inside each Episode, the Agent performs numerous Action before victory or defeat, each time the Agent performs an Action is a Step.

We do not know in advance how many steps there are in an Episode. Thus, to control the convergence of the Policy, we have to adjust the number of Episodes we want.

There is still the last problem with our algorithm. At the beginning of the Training, we do not know anything about the Policy. Thus, all its values are set to 0. However, the Bellman's Equation looks for the maximum of the Policy over a given state. If all the values are 0, the maximum will arbitrarily be always the same. To avoid this bias in the algorithm we ask the algorithm to choose an Action at random at the beginning of the Training, disregarding the Policy.

The probability with which we ask our algorithm to choose an Action at random is called the Exploration Rate. We expect this Exploration Rate to be high at the beginning of the Training and low at its end.

Thus we have to choose three new hyper-parameters.

The maximum of the Exploration rate, between 0 and 1.

The minimum of the Exploration rate, between 0 and the maximum chosen above.

The decay of the Exploration rate, greater than 0. At the end of each Episode, we decay the Exploration rate with an exponential law.

$$E_{t+1} = E_{min} + (E_{max} - E_{min})e^{-decay \times t}$$

Decay Formula, E is a shortcut for Exploration rate, and t is the number of the current Episode

These three new hyper-parameters have to be chosen by hand before the Training. Like with the Learning rate, the Discount rate or the total number of Episodes, if they are poorly chosen they could prevent the algorithm from converging.

Programming the Algorithm

Because this course deals with the Internet and that the main languages of the Internet are HTML/CSS/Javascript, we choose to program the algorithm with these languages only. That means we have to go through some extra steps because this kind of work is usually done in Python3 and the libraries used are not available in Javascript.

All the programs are given in an Annexe Folder.

Website

First, we need to build the Website that will host our experiment. There is nothing hard with this. We just figure out what we need and use HTML5 - index.html - to create the element. To make the Website more readable we use CSS3 - Styles/styles.css - to render the game in a pleasant manner. Then we link each element to a Javascript - Scripts/main.js - to then be able to train and test our algorithm interactively.

On our Website, we will need:

A button to Play the game (by a human player) ;

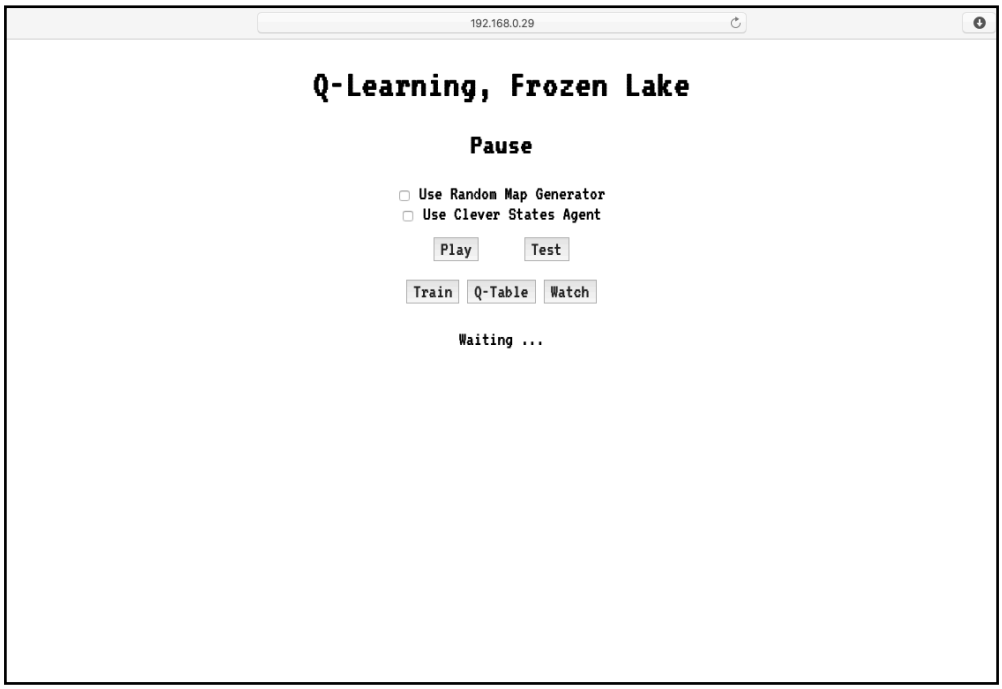
A button to Test the algorithm ;A button to Train the algorithm ;

A button to display the Q-Table ;

A button to Watch the algorithm play the game (this one isn't really necessary) ;

A field to display the text (render the game or display results from training or testing).

We chose to render the game with only letters, layout in a grid, to keep the work easy and to avoid making graphics.



Screenshot of the Website. The main interface is laid out.

Environment

Then, we have to build the Environment. This one is pretty straight forward and just requires you to follow the instructions of the game.

We will work with a preset map commonly used for this problem.

S	F	F	F
F	*H*	F	*H*
F	F	F	*H*
H	F	F	**G**

The map used for playing and training the FrozenLake game. This map is commonly used because it is feasible but hard to succeed by chance due to the position of the holes.

To make our Environment useful, we need two functions.

The first one is the reset(…) function, that restarts the game. It is called at the beginning of every Episode.

The second one is the step(action) function. When this function is called it makes the player perform the given action. It is important to let this function in the Environment scoop to avoid any bias during the programming of the Agent next. Indeed, these functions have to be black-box, knowing what is inside when programming the Agent could lead to an over-efficient algorithm that does not reflect the reality. This function will tell the Agent if the game ended - i.e. if it lost or won -, what is the Reward it earns and what is its new State.

Agent

The Agent is mainly made of two functions.

The first one is the train(…) function, that performs the Training as explained. Thus, it updates the Q-Table at every step according to the Bellman's Equation.

The second one is the exploit(…) function that chooses which action to perform according to the Q-Table. This action is simply given by the maximum of the Q-Table over the current state of the Agent - each action is mapped by an index corresponding to the index in the Q-Table -. This function is dissociated from the train(…) function because it is then needed for training and watching our algorithm.

Hyper-Parameters

After some tests and looking on the Internet what values other people used, here are the retained hyper-parameters.

Number of Episode: **10 000** ;

Learning Rate: **0.1** ;

Discount Rate: **0.99** ;

Maximum of the Exploration Rate: **1** ;

Minimum of the Exploration Rate: **0.01** ;

Decay of the Exploration Rate: **0.001** ;

We also needed to determine what will be the reward of each state. Here, even if there are sixteen states, there are only four different states - S, F, H and G -. We used the following system of reward during the Testing phase.

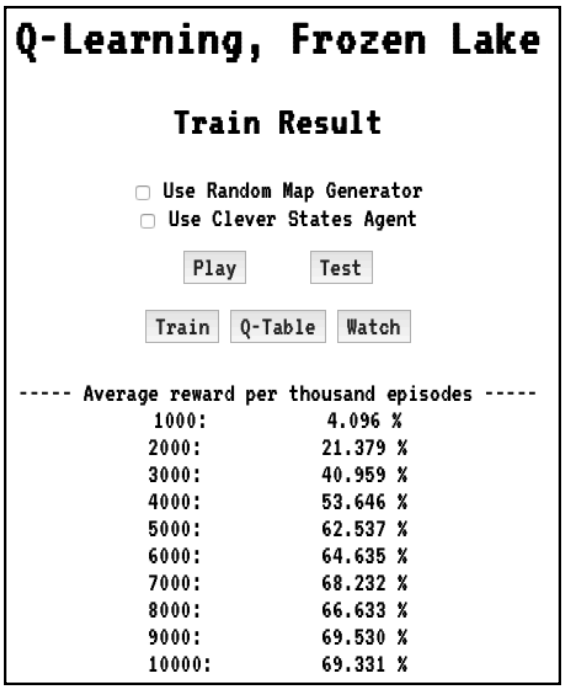
	Reward System
S (normal)	**0**
F (normal)	**0**
H (hole)	**0**
G (goal)	**1**

System of Reward used during the Training.

This mapping means that we only care about going to the Goal. We rewarded the Agent every time it reached the objective of the training.

Result

After the training, we tested our algorithm for 10 000 more Episodes. It appears that it found the Goal without falling in a Hole **73% ± 5%** of the time. The algorithm took less than a second to execute.



Screenshot of the Website just after the Training Phase.

Screenshot of the Website just after the Testing Phase.

Limitation of the algorithm

However, the map was always the same in this problem. Thus, the Q-Table only predicted the direction to choose in a given position, disregarding the potential obstacle.

Thereby, if we change the map at every Episode for the testing, we obtain a rate of success of $\mathbf{15\% \pm 2\%}$. That is way too low to say that the algorithm works on any kind of maze.

We then try to figure out a way to tackle this issue without having to change our algorithm. Indeed, we could use a Deep-Q-Learning algorithm that is much more performant for finding a path in a random maze. However, these algorithms are hard to explain and time-consuming. Thus, we will keep our simple algorithm and try to make it bet the random maze.

In the next section, we explain that we obtain a success rate of $\mathbf{45\% \pm 10\%}$ only by changing our understanding of the game. The uncertainty of the success rate is easily controllable. Indeed, because the algorithm keeps simple, it does not take more than three seconds to execute. Thus, it does not need a lot of time to find the best success rate by doing, again and again, the Training Phase.

Adaptivity of the Project

New Mapping of the Environment

The man problem of our previous algorithm is that we describe the Environment with a complete set of States. That means that every State maps to a different situation in the learning Environment.

Thus, if the Environment has to change, all the training has to be redone.

However, the main advantage of this method is that it is efficient. Indeed, the Q-Tables only describes the Environment, thereby the Agent has learnt by heart how to act. This needs a few numbers of step to converge (towards a good Policy, i.e. Q-Tables).

Here, we want the maze to change from time to time, so the states could not map completely the Environment. If we do so, the Policy will try to describe the new Environment every time it changes and thus will never converge.

We have to think of what are the things that keep being present disregarding the Environment. The initial position of the player and the goal's position are always the same: (1,1) and (4,4).

Then, our player will always have four directions of movement. So, it will always be something in each of these directions. This something could be either the goal, a normal tile, a hole are the limit of the maze (i.e a wall). Thus, we can map our Environment regarding only what are the adjacent tiles of the Player.

Now, let us have a look at the number of states that this mapping implies.

For each direction, we have three possibilities: normal tile, hole or wall. We do not consider the goal possibility because the Goal is always at the same position, thus by reaching the Goal during the training, the needed information will be provided to the Q-Tables without having to tell so.

That makes us a total of $3^4 = 81$ different states. That is much greater than the 16 states we had. Moreover, this only takes into consideration what it is just around the player, thus there is no understanding of the maze possible.

That may seem odd, indeed, we have way more states, and they only cover the nearest tile whereas with our old representation we had fewer states and we cover the whole maze. However, now we have a pattern that can handle any kind of maze.

We could have chosen a different mapping, such as looking a 2-tiles around or looking at the corner tile too or adding the distance from the origin as a parameter. However, all these ideas, even if they may end up with a more precise representation of the Environment, have two main problems.

The first one and the most obvious is that the number of states is becoming too large. Thus, the calculation time too and our algorithm will not be efficient enough. Just as a matter of example, for the ideas listed above, the number of states are respectively 6561, 6561 again, and 1296 (coming from the results of 3^8 and $3^4 \times 4^2$ respectively). We do not have to forget that for each State, we have four values (one for each Action) and that each of the values has to be filled. That means that the Agent has to go through all the possible states and perform and the possible actions. The more states there are, the more Episodes we will need. As an idea, for our previous 16 states, we needed 10,000 Episodes, thus for 6561, we will need more than 1,000,000 Episodes.

The second one is more strict. Even if we add the time to train our Agent with its 6561 States representation, the Policy will never converge. That is because a lot of states represent a very similar situation in the Environment. The value will never stop to oscillate between many limits (one for each unique situation), and thus never converge to the good one. That is a common problem in Machine Learning. We have to take care that the dimension of our training space is not too greater than the real dimension of our space - here the dimension of the training space is the number of states and the real dimension of our space is 16, the minimum number that describes fully the Environment, i.e. the number of tiles).

Random Generation

Now that we have implemented our new system of states, we can focus on the generation of a random maze. The algorithm to create a new random maze in pretty straight forward.

1. Filled each tile with F (normal tile) or H (hole tile) given a probability α, α the probability of being a F and $1 - \alpha$ the probability of being a H
2. Replace the tile (1, 1) by S and the tile (4, 4) by G

3. Check if there is a path of F from S to G. If not return to step 1.

To check is there is a path, a common *Deep First Search* algorithm is used. Basically, this looks for each path starting at S and stop when it finds G.

Now that we have implemented our new system of states, we can focus on the generation of a random maze. The algorithm to create a new random maze is pretty straight forward.

1. Filled each tile with F (normal tile) or H (hole tile) given a probability α, α the probability of being an F and $1 - \alpha$ the probability of being an H
2. Replace the tile (1, 1) by S and the tile (4, 4) by G
3. Check if there is a path of F from S to G. If not return to step 1.

To check if there is a path, a common Deep First Search algorithm is used. This looks for each path starting at S and stops when it finds G.

Before making the training, we have to think about how we will make it. Now the Environment can change, thus we have to change at which frequency we will generate a new maze.

Indeed, if we set this frequency to 0, we do not change the maze at all. Thus, the Agent will only see one maze and the Policy will describe the structure of this precise maze. The Agent will not be able to find the goal in a new maze.

On another hand, if we set this frequency to 1, we will change the maze at every Episode. This may seem a good solution because we want our Agent to learn to move in any maze, however, let us see why this is a bad idea.

To update its Q-Tables interestingly, the Agent needs to go many times in the same situation. However, if the Environment changes at every Episode, the Agent will never find two times the same situation. Thus all the value on the Q-Tables will stay near 0. One good idea may be to update the maze only one a while - every 1,000 Episodes for example, however, the frequency has to be found by hand, it is a new Hyper Parameters).

Some mazes can be harder than others (let us take a look at the two mazes below), indeed because the movement is partly random, some mazes can be almost impossible even for humans whereas others can succeed even if the agent pays at random. Thus, if we generate an extremely hard algorithm, the Agent will keep losing, even if it got a pretty good

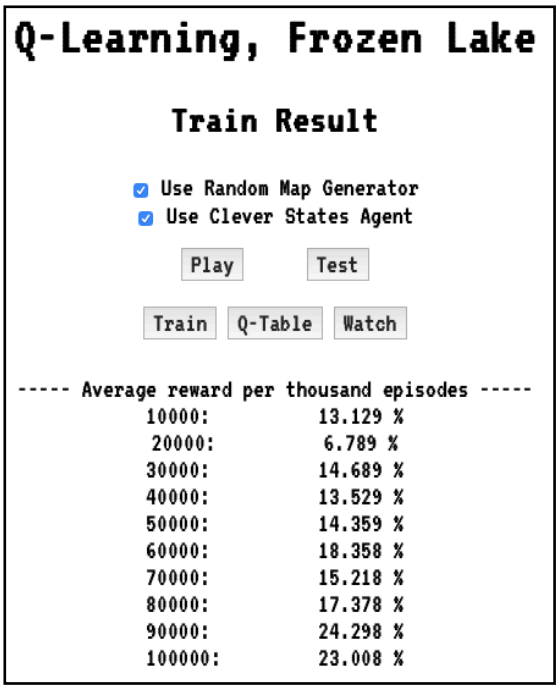
Q-Table. However, because it failed, it will update its Q-Tables to act differently, by doing so, it will diverge from its solution. Therefore, we have to remove the hard maze at the beginning of the training. Because the maze is generated randomly, we cannot test them all. So, to test if a maze is hard, we look at the number of failures of our Agent, if the Agent fails too many times, we skip the maze. The number a fail needed to skip the maze is once again a new Hyper-Parameters.

S	F	F	F
F	F	F	F
F	*H*	F	*H*
F	*H*	F	**G**

S	F	*H*	*H*
F	F	F	*H*
F	F	F	F
F	F	F	**G**

The maze to the left is much harder to the maze to the right even if they both have the same number of holes. Indeed, in the maze to the left, the player has to go in between two holes. However, because the movement is partly random, there is 66% to lose at each try, even if we are taking the good path.

We can now perform our first pieces of training to find the best new Hyper-Parameters, however, when we did the training we found that the Policy never converges (see below). However, this is mainly due to our reward system as we will see.



Screenshot of the Website just after the Training Phase. We see the success went up and down and its mean did seem to have any clear tendency.

A new Reward System.

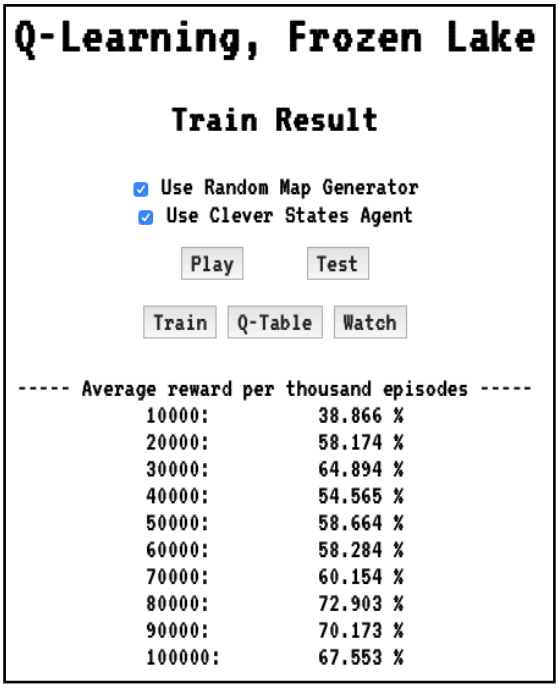
We have to look at what our new States representation means. We are only looking at the adjacent tile, that means that we are focusing on avoiding the hole. We want our Agent to survive long enough to find the Goal. Thus, our old reward system does not fit our purpose anymore, we now have to reward our Agent each step for being alive. In addition, because we are focusing on avoiding the holes, we will now punish our Agent each time it falls in a hole.

It is another common error in Machine Learning to not understand the basic meaning of a part of an algorithm and then mix different algorithms with different meanings into one algorithm. It is extremely important to change every algorithm we took to make them coincide with the meaning we want. If we do not do so, we can prevent our algorithm from converging, and this was what has happened just before.

	Old Reward System	New Reward System
S (normal)	**0**	**+ 0.001**
F (normal)	**0**	**+ 0.001**
H (hole)	**0**	**-0.5**
G (goal)	**1**	**1**

New system of Reward used during the training. Here we can see that a little positive reward is sent to the Agent each Step (+0.001).

The Policy now has a better tendency (see below). However, there is still important oscillation, this is due to the change of algorithm. Indeed, after each change, the Agent will endure some failure because it has begun to learn by heart the previous maze. That explained why we have this sawtooth tendency.



Screenshot of the Website just after the Training Phase. We see the success went up and down but it is meant to tend to go up.

Training

We can now choose new Hyper Parameters.

We remake a new maze every 1,000 Episodes. We call this number R.

We choose to not look at the difficulty of the maze. When we low R, the difficulty of the maze tends to have no influence, mainly because the Agent does not have the time to forget what it has learned from its previous training in only 1,000 Episodes, even if it is 1,000 failures. At first, we had chosen R to be 5,000, so by doing so, we were obliged to remake a new algorithm if the Agent failed 500 times during the 1,000 first Episode of the new maze. However, this was not convenient to implement and our new choice of R appears to be more efficient for the convergence of our Policy.

Because the number of states has grown up to 81, we update the number of Episodes to 100,000. That is 10 times larger, simply because we used to have 16 states, that is roughly 10, and now have 81 states, that is roughly 100. Thus there is a ratio of 10 between both.

We tried to take another number of Episodes created than 100,000 but that did not change the final result. That is because of the Sawtooth tendency of our success rate, after too much training, the drop resulting from a new coming maze is as large as the gain from the learning on this new maze. Indeed, the gain from the learning of a given maze is getting down after each new maze, simply because the Policy cannot converge as fast as before - the

convergence isn't linear, it is rather defined recursively because of the initial formulation of the Policy, thus the slope tends to decrease -. However, the drop after each new maze is almost random, so the range value it can have does not change a lot - even if it depends from the current number of Episode, the dependency is too low -.

We then try to train with both our reward system, to prove that the second one is more efficient - because it has the same purpose than our state representation -. The training took around 5 seconds.

To perform the testing phase, we generate 10,000 random mazes and make our trained Agent play in each of them. Then we calculate the rate of success, i.e. the number of times it reaches the goal over the total number of trials.

When we use the old reward system we have a success rate of **15% ± 5 %** . As a reminder, we had **15% ± 2 %** with our previous algorithm. Thus the only thing that changed is the uncertainty because the training phase is really fast, we can obtain an algorithm with our new method - i.e. an Agent that succeeds 20% of the time versus an Agent that succeeds 17% of the time-. However, the difference is too low for saying that we have a real improvement.

Then, we use the new reward system and obtain a success rate of **45% ± 10 %** . Because the training phase is fast, we can then obtain an Agent that succeeds 50% of the time. That is much better.

Conclusion

In this report, we have seen an extremely basic method of Reinforcement Learning. Then we had applied this method to a static Environment and a changing Environment. We adapt our simple algorithm to multiply by three its success rate, up to 50% of success, in a changing Environment. Due to the difficulty of the game we were playing, a human-success-rate wouldn't be higher than 60%.

These methods could be applied to more complex Environments like LAN or the Web in a common problem in networking or in web recommendation, or others as we have seen.

However, we have only used the easiest method to tackle this kind of problem. A lot more algorithms can succeed in a very difficult algorithm nowadays. We do not have to use the Neural Network for example in our report, even if our Agent learns to act differently according to the situation.

This report was the first view into the field of Reinforcement Learning and hopes to bring an idea of all the innovation that it could bring to the Web and the way we see it and construct it.