Chapter 1 – part 2

**Artificial Intelligence:**

**Fundamentals of**

**REINFORCEMENT Learning,**

**Introduction to Q-learning**

Policy vs Plan, Penalty

Q-Learning, Temporal Difference

**1.8 Policy vs Plan**

Here we'll discuss about POLICIES versus PLANS. Now we consider our environment/world with stochastic i.e. it's in nondeterministic search.

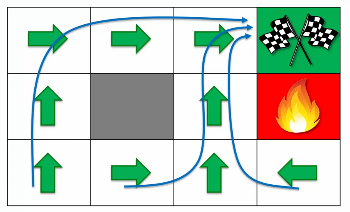
* You're just not getting through the maze but also facing some random factors and you need to be prepared for it.
* Following is our **MDP-framework**, which is actually a modified Bellman Equation.

|  |  |
| --- | --- |
|  |  |

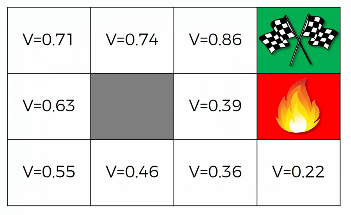
* The value of any state as is the *maximum across all actions* an agent could possibly perform in that state.
* And the maximum was taken from the reward that the agent will get by perform action in state and added to a discount factor multiplied by the *expected value of the new state it will be in*.
* So let's have a look at this our example of the maze. Previously we're dealing with the Deterministic Search, where no randomness/uncertainty was present: if the agent wants to go up, it will definitely go up (no uncertainty).

|  |  |
| --- | --- |
| * It was all pretty straightforward and we called it a PLAN. And it represented with arraws: |  |

* Following are very straightforward routes that the agent will take whenever it start on those blue line.



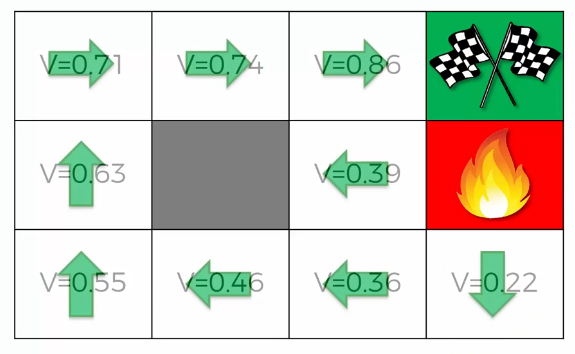
* In terms of PLAN, we know exactly what to do next, what are the steps, we had a start point and a goal. So we can plan them out.
* **Policy:** However in Nondeterministic Search we ***don't have a plan anymore***. Because, whatever plan we prepared for, might not happen, it's not under control. There's so much now randomness going on.
* For example, if you want to go up but actually it takes you down. That's not part of your plan, and that's why it's not called the planning anymore.
* So now we want to *calculate* the *values* again but this time *taking care of the* ***RANDOMNESS*** (stocasticity) and we got the following result.



* Let's compare to what we had previously. By the way these are *not exactly the correct values*; these are just for our learning purpose.

|  |  |
| --- | --- |
| Deterministic Search (PLAN) | Nondeterministic Search (Policy) |
|  |  |

* Now, consider the First Square, next to the Goal: previously we had ***1.0*** there, but now we got ***0.86***.
* This is because, say we want to go to the right (to the goal), but there is a possibility that we *hit upward* and back to the same-state. Since we have a *discount factor* () the *value* reduced.
* So there is no 100% probability to go to the right.
* This is applied to all other squares of the maze.
* Near Firept: Now consider the box next to the Fire-Pit, previously we got **0.9** there but now we got **0.39** it has dropped substantially. Why is that?
* Because if you want to go ***up***, there's a 10% chance of actually ending up in the Firepit and getting ***minus one (-1) reward*** and *end of the game*. Also there's a 10% chance of hitting a wall.
* So this is a very bad state to be in. That’s why in non-deterministic search we've got **0.39** instead of **0.9**.
* Similarly we've got **0.22** for the square below the firepit and **0.36** on the left, because there is a chance to end up in firepit and having -1 reward.
* So in this case the arrows changed:

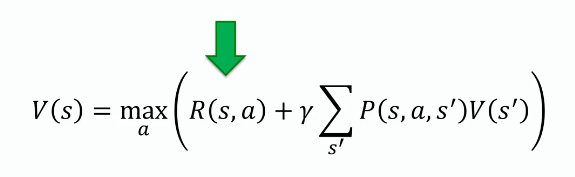


* Notice the arrows near the *firepit*, both are towards the wall. Agent is *not going* to the *firepit*, but there is a chance to *go up* and *move* *away* from the *firepit*.
* It can *bounce* the *wall couple of* *time* and start to moving away from the *fire-pit*. It learns through its failure.
* Hence the routes are changed:

|  |  |
| --- | --- |
| * That’s what the Policy vs Plan is.   The AIs coming up with very different kind of decisions.   * Sometimes in the real world the ideas from AI can be totally different that even a human cannot generate. * That's exactly what happened in Google Alpha-go it came up with some moves that humans had never played in *3000 years* or humans were *not used to play*. |  |

**1.9 Adding a LIVING PENALTY (Shortest Route)**

* Here we'll discuss the living penalty. Following is our modified Bellman Equation, we applied MDP and used expected values. Now we want to modify this equation more.
* Now we want to **modify** the **Rewards** (it's about to finding Shortest Route):



* Now notice that in our maze example we have rewards that are given to the very end (either finish-line or fire-pit).
* But the *reward* can be given *throughout* the *journey*, for taking *different actions*. For example, in some *video game* you get *points* by *killing enemy* or *stealing cars* etc. The same thing we can do for our agent in reinforcement learning.
* We can give rewards to the agent for taking different actions.
* That's why we're going to introduce something similar into our example: a *reward* that is *continuously* *given* to the *agent* throughout the *game* (not just at the end).

|  |  |
| --- | --- |
| * Right now we only have a reward **+1** at the final-tile (finsh-tile) and reward **-1** at the other final tile (the firepit). | * But now we're going to add rewards in every single tile. |
|  |  |

* We'll add a very small reward, which will be minus 0.04 (**-0.04**).
* Living Penalty: It's **negative** so that ***every time the agent moves he'll get a negative reward*** and that's called a **living penalty** because no matter where he goes he will *always get this negative reward* except for the final tiles because that's the end of the game.
* If the agent start from any of this tiles, he's not getting the penalty.
* It only gets this reward (penalty) when he moves/enters to another tile.
* Even if he comes back to his previous tile he'll get another penalty.
* More moves, more penalty: The longer the agent walks around the more it get the negative reward (penalty).
* Minimize Penalty: Therefore it tries to finish the game as quickly as possible, by taking minimum steps/moves. So it tries to minimize its penalty.

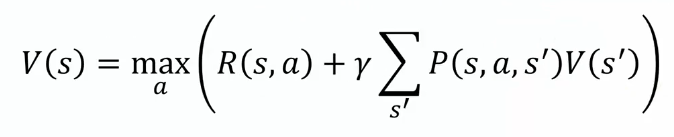
|  |  |
| --- | --- |
| * How LIVING PENALTY changes the POLICY: Let's have a look at how the agents policy is going to change depending on what value we set for this reward (penalty). * So here are four environments and in each one we're going to explore a different reward. * We're not going to do the calculations. We're just going to project the results. |  |

|  |  |
| --- | --- |
| 1. Before getting into any state, reward is 0.  * So this is our original policy. And remember we had these two very interesting and weird a decision by the agent to live for as long as he likes (not getting the fire-pit but taking too many steps/moves). We'll observe these two tiles for different reward-cases. * Since we are not setting any penalty, it can moves here and there as long as he wants. |  |
| 1. Now let's see what happens if we add a negative reward for making a step.  * And notice that instantly these two weird -moves changed. * Now the agent doesn't want to jump into the wall. He is more likely admit ***10% risk*** to getting into ***the fire-pit*** but with ***80% chance*** to going ***forward***. |  |
| * Let's consider the tile between the wall and the firepit, so if it hit the wall he takes an action and end up its previous step and getting **-0.04** penalty. There is also ***80%*** chance to moving forward and getting another ***-0.04*** penalty. * So there is *lots of penalty* it will get *by hitting the wall*. To *minimize* the penalty it will then *stop hitting the wall* and tries to *move forward* with 80% chance. * Same thing goes for another tile below the fire-pit, if he keeps hitting the wall he will accumulate this negative reward and you'll see that the expected value of that approach of hitting to the wall is worse than taking the risk of going forward and actually ending up in the firepit. * So he changes his decisions in these two blocks/tiles and move forward and left (tile below fire-pit) even there's a risk of entering the Firepit. | |
| 1. In the 3rd environment we increase the living Penalty **-0.5** (from **-0.04**).  * Notice that the tile below the wall changes direction to the right (and upward next tile), because to minimize **living penalty** this is the shortest route. * Taking the safe path makes more Penalty than the shortest path. |  |
| 1. Finally, we increase our penalty to **-2.0** (which is more than jumping into **firepit**).  * To minimize the penalty agent jumps directly to the **firepit**. |  |

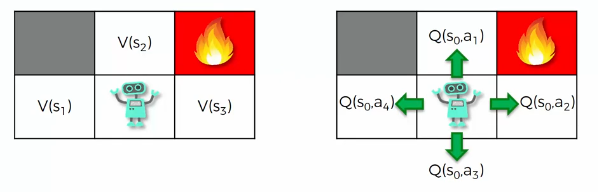
* So we can see that we are getting different results for different Living Penalty. The agent is going to select different policies.
* That’s how reward in Bellman equation is given throughout the game.
* However the rewards ***not need to give every single state***, it can depends on the ***environment itself***, this reward can be given to the agent for ***certain states*** (not in every single state).
* *For simplicity's sake we gave reward for each state in our example.*

**1. 10 Q-Learning Intuition**

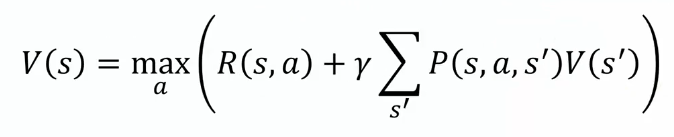
* What is Q learning:
* "*Q* " refers to the *function* that the *algorithm computes* – the ***expected rewards*** for an ***action*** taken in a given ***state***.
* Quality: In Q-learning, 'Q' stands for quality. It represents how useful a given action is in achieving future rewards, which is used to create a map *system of state* and *action* to maximize *expected rewards*.
* Q-learning is a Model-Free Reinforcement Learning Algorithm to learn the value of an action in a ***particular state***. It does not require a model of the environment (hence "model-free"), and it can handle problems with stochastic transitions and rewards without requiring adaptations.
* For any Finite Markov Decision Process (FMDP), ***Q-learning*** finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state.
* Q-learning can identify an ***optimal action-selection policy*** for any given FMDP, given *infinite exploration time* and a *partly-random policy*.
* "*Q* " refers to the *function* that the *algorithm computes* – the ***expected rewards*** for an ***action*** taken in a given ***state***.
* We have the following Bellman equation; which is a recursive function, we've modified it with lots of components.



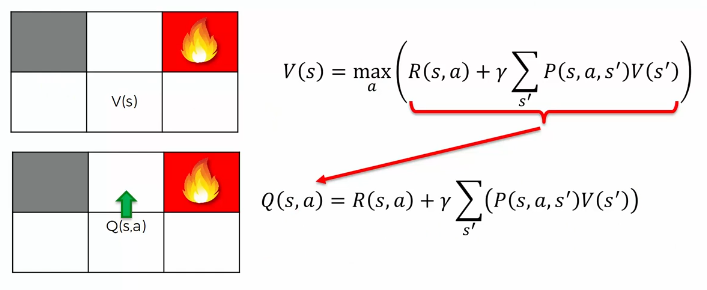
* **Reward:** Now it is given throughout the game.
* **MDP:** also we added stochasticity to our environment and applied MDP to our equation.
* Where is Q: The question is there is no letter Q in the equation or in the process. **Why we call it Q-learning?**
* So far we've been dealing with ***values***  of being in a certain ***state* i** and now we're going to implement ***Q*** in that.
* So here we've got two examples, on the left is Value-of-State approach and on the right is Q-value approach.



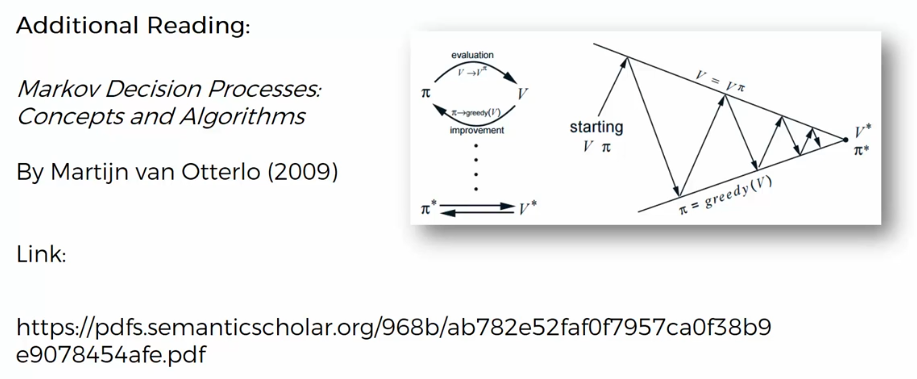
* We have a MARKOV DECISION PROCESS (MDP), so doesn't matter how the agent got there.
* Agent tries to make the *optimal decision* about where to go, based on the *current state*
* All the *future states* come from *present* not from the *past*.
* There's three options: states
* Based on its experience the agent has calculated the ***values*** in these states
* Then it's going to use the Bellman equation, to make decision.
* On the right side we've got **Q-Value approach**, which is modified a little bit.
* We're going to take the same exact concept/problem but here instead of looking at the value of each state , we're going to look at the ***values of each ACTION of each STATE***, Eg: for being at state , we are going to look at .
* So we're not going to use the letter anymore, we're going to use . '**Q**' stands for quality. It represents how useful a given action is in achieving future rewards.
* So Q here represents the ***quality of the action*** rather than just a ***value of a state***.
* What is Q exactly? How to find which action is more useful?
* Q is a metric (a formula), telling us: how do we quantify these actions and then we can compare them. And that is exactly what Q is.
* In above example, in a state the agent can take four possible actions (go up/right/left/down) and are the different ***qualities*** or ***Q-values*** of these ***actions***.
* And based on these actions, there's going to be a formula which tells us the quantifiable value of each action, which we're calling the Q-value of that action.
* Now let's have a look at how we going to derive this Q-value.
* Since certain actions lead to certain states, there must be some sort of link between them.
* In Value-of-State approach, we've already determined how to calculate it and we know how to use the Bellman equation in very different environments with lots of different complications.
* We'll leverage that knowledge to understand how to calculate the Q-value.
* Since the environment is same in both approaches, so there must be a *link* between these *two approaches*. Because the environment doesn't change depending on what approach we use.
* Therefore this ***Value-of-State approach*** and ***Q-value approach*** should always give the ***same result***. And that's another reason why these two should be linked.
* **How do we relate V-value and Q-value?**
* Here we have the **V-approach** (Value-of-State approach) and **Q-approach** (Q-value approach).
* **V(s)** depends on state **s** only and **Q(s, a)** depends on state **s** and action **a**.
* **V(s)** is the maximum of all possible action **a**. Defined by the following Bellman Equation.



* **Q** is going to be defined by the above Bellman Equation.
* Let's say the **agent** wants to go **up** from its current state **s**.
* **Firstly**, there will be a reward .
* **Secondly**, there is now quantifiable metric:
* It could move upward or left or right or hit the wall.
* But wherever he ends up, there's already a quantified metric for that state he'll in.
* And that is actually the value of that state for being new state .
* Since there are different possible states, we have to look at the expected value of the possible new states that it will be in. So we have to add .
* We also going to apply the discounting factor .
* So following will be our Q-value for this for performance action:
* Notice that
* Here in we will get the maximum across all possible actions.
* But in we're defining "what we will get by taking a certain action".
* So the value of a state is actually the maximum of all Q-values .
* For Example, our agent currently has **4** Q-values for move *upward* or *left* or *right* or *hit the wall*. So will be the maximum of all those four Q-values.



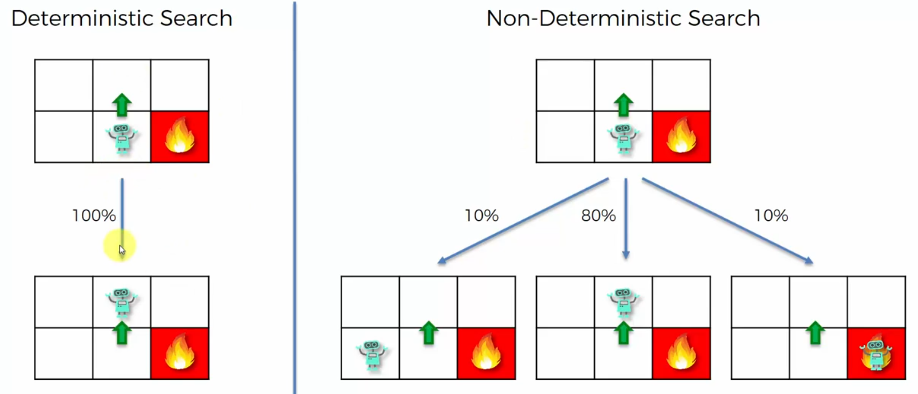
* **Q-value** is a quantified metric of performing an action and
* **V** is the maximum of the possible results of the four actions that it can perform.
* We can modify the formula further, because is a recursive function. We can replace by . Where are possible new action for corresponding new state .
* That’s how and are linked.
* And now we have recursive formula for Q-values.
* So that’s the powerful Bellman equation for Q-values, which we can now apply and let our agents learn how to beat the environment.
* Additional reading: Following paper is called **Markov Decision Processes: concepts and algorithms** by Martijn van Otterlo (2009). Here you can read in a bit more detail to understand all the basics behind Q-values.



**1.11 Temporal Difference**

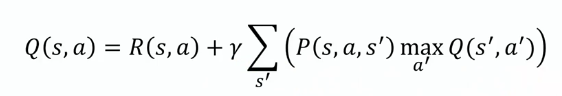
**TEMPORAL DIFFERENCE** is the heart and soul of the Q-learning algorithm. Everything we've learned so far comes together into play inside Q-learning, when we apply the Temporal Difference.

* Remember the DETERMINISTIC versus NONDETERMINISTIC search.
* In deterministic case when the agent wants to go up he definitely goes up. But in NONDETERMINISTIC case if he wants to go up there's a 10% chance he'll go left and 10% chance to go right and an 80% chance to go up.



|  |  |
| --- | --- |
| Deterministic Search (PLAN) | Nondeterministic Search (Policy) |
|  |  |

* Also Remember, in deterministic search it is easy to calculate Value of the state. We simply calculate them by using the Bellman equation and get the exact values. There is no stochasticity.
* But for the Nondeterministic Search it's very complex to calculate (and for learning purpose we used some fake/made-up values).
* Why is it so much *harder* to *calculate* these *values* in the *nondeterministic* example?
* Because when the agent moves, for instance to the right, there are chances to move up or down or left.
* So in order to calculate the value for moving right, you need to know other values of moving up or down or left.
* So there's a lot of ***recursion*** happening here and therefore you cannot just decide to define what these values are. Moreover, this ***recursion*** is ***nondeterministic***.
* Hence it is subject to chance and many iterations that the agent taking certain path and suddenly it changes the path and end-up in the fire-pit and then the value of state drops/changes.
* So there is some stochastic randomness to this whole calculation, and these values are all interlinked.
* And the **environment** consist **Markov Decision Process**.
* So that's where all this comes together and here we're going to introduce the concept of the TEMPORAL DIFFERENCE; which will allow the agent to calculate these values.
* And since we are moved from V-values (nondeterministic) to Q-values, we'll discuss TEMPORAL DIFFERENCE using Q-values.
* Temporal Difference (what is it ?): At this point we have following Q-Value-Bellman-Equation.

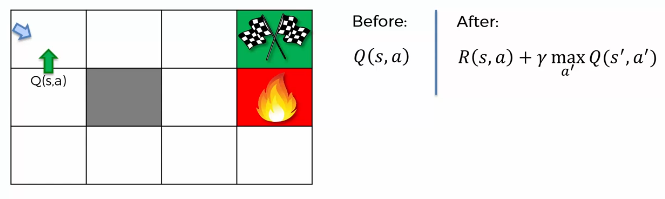


|  |  |
| --- | --- |
| * For simplicity, we get rid of Stochasticity (randomness) from this equation. We're going to rewrite it in the deterministic search (without the expected value and probability): * But remember, our goal is to apply it in Non-deterministic search. |  |

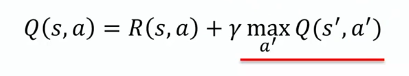
* Following is a blank state of the maze, we don't have any Q-values. Let's just look at one of the states or one single cell.

|  |  |
| --- | --- |
| * We have an action of going Up, we have a **Q-value** . (We have a Q-value that we calculate. For simplicity, we're not illustrating anything. We're just keeping a blank for simplicity's sake.) * We assume that the agent has been walking around for some time and let's say hypothetically somehow he's calculated this Q-value of going up. |  |

* Currently the Blue Arrow points that the agent is sitting in this cell.
* And now he needs to make a choice. He knows the value of this action of going Up.
* Before: Let's say this is a "before value".
* The reason for that is, he hasn't taken action yet, he's still in the cell and before taking any action, the value here is .



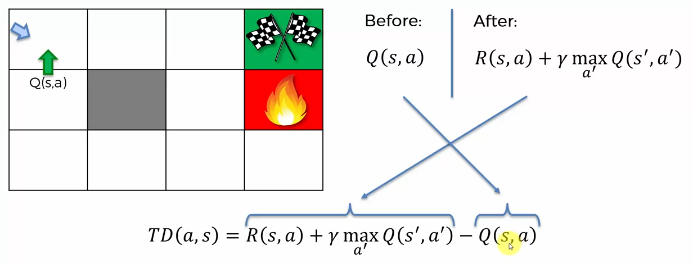
* After: Let's say agent takes an action according to best choice,he decides to move Up.
* Now comes "after". So after he's taken action we can measure this after-value, using our equation. Where, is the new state.
* Recalculation Happening: Now notice that, in the following equation, in the ***left side*** we have the ***Before-Value*** and in the ***right*** ***side***, we have the ***after value*** (just calculated).
* We had value before that action. And then we've calculated this metric after the action.



* So before we knew value, which the agent has calculated through iterations. Let's say this value then stored in agents memory.
* After the action is being performed, we know what reward the agent actually got.
* And we then calculate .

So in essence we're kind of recalculating this value but now with *new information*, the *new reward* that we've got.

* Temporal Difference (The Definition): The temporal difference is the difference between the "after-value" and "before-value" . The temporal difference is denoted by .
* Which is actually following equation, but there is a difference: the value is the "before-value", which is calculated by the agent by moving through the maze (environment). And is the "after-value" which is calculated from ***reward*** & ***Q-value*** for the ***next step***. So there is a difference between the left-side and right-side, and this difference is the temporal-difference .
* is the difference between these two:
* After value: its calculated afterwards, by going to a new state.
* Before-value: the previous which the agent had stored in its memory.



* **The question is:** are After value and Before-value different?
* Ideally they should be the same. Because is the formula for calculating .
* But, is something that we have from **empirical evidence**. It is the value, that we have from just *going through* the *maze* many times (iterations).
* It's not related to the current iteration.
* It's something that we came up with our previous iterations, going through the maze.
* Whereas is something we've calculated just now (current iteration)
* So there is no guarantee that and going to be the same.
* Because there is randomness exists in the maze.
* For example: could have been calculated under certain random events.
* And is being calculated under different random events.
* **How Temporal Difference is used to train the agent:**



* The reason it is called the temporal difference is because you're basically calculating the same thing, and are basically same.
* But the only difference is **Time**.
* is the value we had previously and is new current value. There's been a shift between them in time.
* And how can we use this difference to our advantage?
* What will happen if we don’t use the "Before-value": If we get rid of ***old value*** , and just use the ***new value***, that would not be smart.
* The reason is that in our *environments* random events can happen.
* And what if our old was something that consistently happens like **80%** of the time.
* And then this new one happened due to randomness. Which has only **10%** or **20%** probability of occurrence.
* So taking care only the new value creates problem, we are replacing something that has ***80% chance*** with something that happens only ***10%*** or ***20%*** percent of the time.
* That wouldn't be the best approach to follow, and that's exactly why we don't want to completely change the Q-values.
* We will change the **Q-value** step by step, a little bit by a little bit. And that's why we're going to use this **Temporal Difference** in a specific way. So here is the formula: Where basically we're taking this Temporal-difference and adding it on to our previous Q-value.



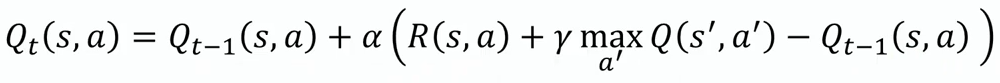
* **left side** is the **new Q-value**, on **right side** we have our **old Q-value** which is updated by adding .
* Here is the **learning rate**. That's a new parameter that indicates how quickly is algorithm learning.
* We re-write those equation in a different way, (something like difference equation):

|  |  |
| --- | --- |
|  | is the **learning rate** |

* We just adding time-steps, and to these formulas.
* is *current Q-value* is going to be equal to *previous Q-value* plus *temporal difference times Alpha* .
* This formula here is the **Heart And Soul** of the **Q-learning algorithm**.



* This is how the Q-values are updated.
* When the agent takes action he'll get a reward and he'll end up in a new state.
* It can calculate the Q-value of that move that it made from .
* It can calculate the Temporal Difference by subtracting old Q-value.
* And it will get the new Q-value from the above formula, using learning rate and Temporal Difference.
* However we can also rewrite above two equations into a single equation as follows:



Now if then , as results has no effect.

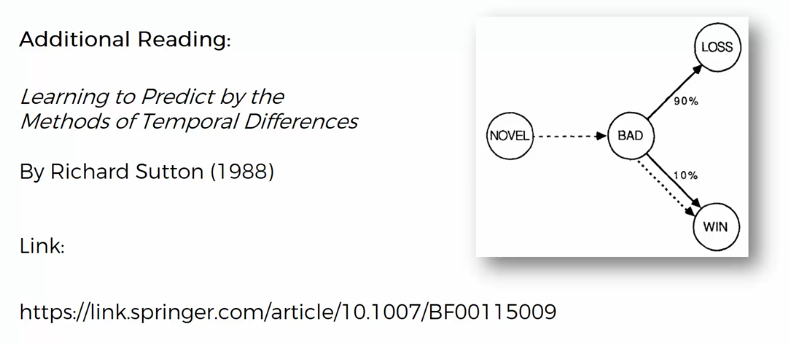
Also if then again loses its effect.

* And that’s why needs to be in between **0** 1nd **1**, to allow you to learn slowly step by step, as the agent goes through the maze and the Q-value updates step by step.
* At some point hopefully the algorithm will converge.
* Then the temporal difference will start becoming closer and closer to zero, and we get , but this time we applied Temporal-Difference and .
* It means new Q-value is equal to previous Q-value. And the agent learned well this time.
* When your Temporal-Difference **converges** to **zero** means your algorithm has CONVERGED and it's not really necessary to continue updating Q-value.
* After getting above equation, it makes sense:
* Why we have the Bellman equation. Not only how it represents the Q-value but also how the agent updates its values and finding exactly what is going on in that environment so that it can come up with the optimal policy.
* **Continuously changing Environment (special case, more complicated):**
* In this case we need to continue updating Q-values.
* If the environment is constantly changing, not just having some randomness/stochastic events in it, but also the environment itself is modifying/morphing/changing with time.
* So the agent ***continuously needs to learn*** because it's ***not possible*** to ***learn completely*** in that environment.
* There is no fixed optimal policy because the **optimal policies also changed with the environment** all the time.

In that case you will need to continue calculate the TEMPORAL DIFFERENCE and update the Q-values.

* Additional reading:

And if you'd like to learn a bit more about temporal differences then a very popular paper is "LEARNING TO PREDICT BY THE METHODS OF TEMPORAL DIFFERENCES" by Richard Sutton of 1988. You can also read his books.



**1.12 Q-Learning Visualization**

**A good source to learn AI.**

<http://ai.berkeley.edu/home.html>

<http://ai.berkeley.edu/reinforcement.html>

Download the zip file

Download the web page using:

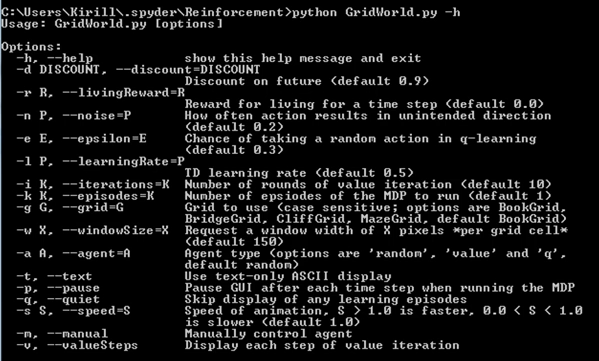
<https://www.web2pdfconvert.com/>



Here we're going to have a look how an AI actually going through that maze in real time. It's going to use Q-learning to navigate its way and find the way out. We'll observe what happens to the Q-values, how the agent finds the policy and so on.

* We're going to be using some materials provided by the Berkeley University. We're going to look at PacMan projects. We need to download the zip archive.
* Then we're just go into Python. After we've extracted all files we could just launch it over python. There are some parameters that are involved in this whole world, read the descriptions.
* Manual mode: Try to launch it in manual mode. Run the GridWorld.py with "-h" (help) command, the all options will be printed on the screen
* By moving the agent we are taking an action. There is also some randomness.
* Now, some random movement will happen, for example if we click "up" agent will go "left".

So that's what it looks like manually controlling the agent.



* AI mode: Now let's apply the AI to this and let it go through.



* We need to add some Parameter.

**-r** : reward

**-k** : number of iterations

**-a** : agent type. Random/value/q

**-s** : means speed

**-d** : is discount

|  |  |
| --- | --- |
| * We already know these kind of parameters from our intuition lectures. Now let's run this.     we do like 10 iterations should be enough.  type of agent we want a Q.  we avoid to set speed  let's keep discount at zero point   * You can see how the agent is exploring, how the Q-values are being updated in highlighted squares. |  |

* Click Right to see the policy that he came up with. Even through just 10 episodes he's already got a good policy.

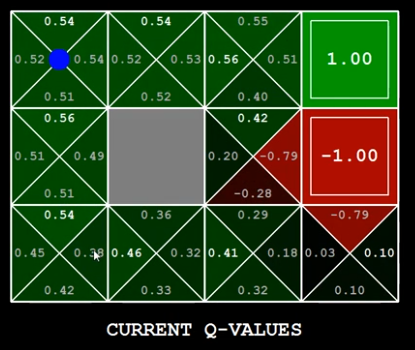


**python gridworld.py -a q -i 20 -k 1000 -s 10**

* Note: Old python 2.x version doesn't work properly. Instead use new 2022, **python 3.10** version from:

[**https://github.com/aliasad059/intelligent-pacman**](https://github.com/aliasad059/intelligent-pacman)

* NOTE: Need to install **python 2.7.14 (2017 version, 64 bit)**. After install rename ***python.exe*** to ***python2.exe*** (if **python3.10** < is default python).
* For many more iterations the agent got more information, more opportunity to experiment and to actually build out the optimal policy.



* You can see **each box** is **divided** into **4-triangles**, representing **Q-values** for next four boxes (up/down/left/right).

So that’s how the q-learning agent learns from the environment.

Next we'll dive into Deep-Q-Learning.