Learning in Agents and Multiagents Systems

Ricardo A. Téllez

Version 0.1

Part One: Autonomous Agents and Learning

1. Learning in agents

An agent learns **behaviors**. Learning a behaviour consists of learning how to interact with the world. To learn behaviors, an agent can use **reinforcement learning**.
There are two types of behaviors:

- **Deliberative behaviors**: these are behaviors as a result of a complex reasoning. These kind of behaviors cannot be learnt because actions are calculated by means of an algorithm. The algorithm is one given by the designer of the agent.

- **Reactive behaviors**: in this case the agent obtains what he has to do from a pool of possible actions. It is a correspondence between situations and actions to perform. The agent learning consists of learning the pool of actions for every situation.
  The behaviors learnt must be stored some way in the agent memory. There are different ways of doing this storage:
  1. Look-up tables: it is a table containing direct situation-to-action entries. The problem is that for every possible situation an entry is needed (even when situations are similar). This method has a bad generalisation procedure.
  2. Neural networks: they generalise very well and are very compact but they have other problems like forgetting things or the impossibility of learning different things.
  3. Rules: a group of rules if-then. Problems with collisions. Obtaining the rules is difficult.
  4. Finite automata: every node is a state of the agent and arrows represent actions. It is compact and readable, but they can only represent behaviors given by regular languages.

By learning it is meant to show to the agent how to reach a **goal**. There are mainly two types of learning:
- **Supervised learning**: the agent is trained with a group of examples that somebody provides. The problem with this approach is that is is necessary to have a lot of examples to show to the agent, and it doesn't allow the robot to find the best possible solution.
- **Reinforcement learning**: the trainer says to the agent if he is performing well or badly. It is a **global** valoration. The main advantage of this approach is that it is not necessary to know a priori which is the behaviour the agent must learn (the agent himself will find the behaviour required for the job, and will find the best of all the possible ones). This method is complex and requires the agent to solve several times the problem in order to acquire the desired behaviour.

On both types of learning, the goal to achieve by the agent is embedded on the feedback that we give to the agent.

2. Reinforcement learning

From [1]:
"*Reinforcement learning is learning what to do -how to map situations to actions- so as to maximize a numerical reward signal. The learner is not told which actions to take, as in most forms of machine learning, but instead must discover which actions yield the most reward by trying them.*"

2.1.Introduction

Under the reinforcement learning paradigm, a system learns while interacting with its environment in order to achieve a goal, but without creating a model of the world. It tries to learn a behavior and it must be the best of all the possible behaviors. Reinforcement learning is not defined by characterizing learning methods, but by characterizing learning problems.

Differences with supervised learning:
- We don't say to the agent which is the correct answer. We only say if he is going Ok or not (evaluation).

- No yes/no possibilities. It is, it cannot be deduced that, when there are only two possible options, if one option is wrong, the other will be right.

Restrictions to the system:

- Actions and situations are produced at discrete times.
- The pool of suitable actions for the agent is finite and discrete.
- The pool of possible situations is also discrete.
- The reinforcement value is a real value. It can be zero (but in that case is not informative).
- It is not necessary that the agent remembers anything (previous steps). The transition from one state to another only depends on the actual state and the action taken. This kind of processes are called **Markovian processes** (decisions and values are assumed to be functions only of the current state)
  A reinforcement learning task that satisfies the Markov property is called a *Markov decision process*, or MDP, and it enables us to predict the next state and expected next reward from knowledge of the current state.

How it works:

The agent and the environment interact at every time step t=0,1,2,3,... At every step *t*, the agent is on a determined state s_t, and he selects an action a_t to perform in response to its state. As a consequence of its action, the agent receives a numerical reward r_{t+1} and finds himself in a new state s_{t+1}. The process is repeated indefinitely.
At each time step, the agent implements a mapping from state to probabilities of selecting each possible action (that's the policy)

2.2. Definitions

- *Policy* : it says for each state the group of actions that can be performed (are available). This is the behavior that the agent must learn and is the core of a reinforcement learning agent. Policies are derived from the *value functions*. Policies can be of two types:
  - Non-stationary: its evaluation depends on the present state.
  - Stationary:
- *Model* T(S,A,S') : probability of transition from the state S to the state S' taking action A. It says how actions modify states.
  Models are used for planning, it is, the way of deciding on a course of action by considering possible future situations before they are actually experienced.
- *Values* : they are used to construct policies. Those are derived from the *value functions*.
- *Value functions*: they indicate the goodness or badness of a state. They are constructed in function of the policy. They are the following:
  - V (S): this is called the state value function. It is an estimation of the long-term reward obtained if been at state S, the agent follows the policy. **It estimates how good it is for the agent to be in a given state.**
  - Q (S,A): this is the action value function. It is an estimation of the long-term reward obtained if after been at state S and taking action A, the agent follows the policy. **It estimates how good it is to perform a given action in a given state.**
  - (S,A): it is the probability of taking action A been at state S following policy.

The notion of *how good* is defined in terms of future rewards that can be expected

- *Reinforcement function or reward function or return* : is a signal given to the agent, in order to score his performance, and defines the goal in a reinforcement learning task.
  It can be also seen as a map of each perceived state of the environment to a single number (the **reward**), indicating the desirability of that state. This function indicates what are good or bad events for the agent.
  It can be of two types: short-term and long-term.
  - The short-term reward (or immediate reward) is the reward obtained by the agent when he changes state. It usually depends on the destination state, not on the origin state.
  - The long-term reward depends on all the short-term rewards the agent is expecting to receive. It is the sum of all the short-term rewards from the present state up to the final state that the agent *thinks* he will receive. The agent will always try to maximise his long-term reward, not his short-term one, so he will learn the behaviour that maximises his long-term reward.

$$R_t = \sum_{k=t+1}^{infinito} r_k$$ been R_t the reward **expected** at step t

Sometimes is not possible to compute the long-term reward because it is composed of a very long sequence of short-term rewards (even infinite). Then it is possible to define a **finite horizon** T that will specify the number of states taken into the sum. The number T depends on the problem to solve and generates new problems discussed later.
Another possibility is to introduce a decay factor into the sum of all the short-term rewards, called the **discount rate**.

$$r = \sum_{k=t+1}^{infinito} \gamma^{(k-1-t)} r_k \qquad with \quad 0 < \gamma < 1$$

The gamma factor expresses that every short-term reward losses importance with time.
Summarising:
The long-term reward is derived from the short-term reward. Posibilities:

- Addition of all the rewards obtained
- Sum of a number of steps behind (finite horizon). This has the problem of non stationary policies.
- Weighted addition of local rewards. In this case, future rewards are weighted so the summitry will be bounded. The policy will be **stationary**.
- Weighted with finite horizon.

The difference between the reward function and the values functions is that the first one represents what is good for the agent in an immediate sense, but value functions represent what is good for the agent in the long run.
Aplication examples: pole balancing, the mountain car problem, the acrobot problem, backgammon or the elevator controller.

Reinforcement learning is specially good on undeterministic domains, on which chance plays a big role (like for example backgammon).

3. Finding optimal policies

An agent doesn't learns behaviors directly. It learns value functions and then policies from those functions. Solving a reinforcement learning task means then, finding a policy that achieves a lot of reward over the long run.

3.1. Main relations and formulas

The two value functions are:

- The state value function V^{Π}

  This value is calculated from the reward as follows (why?): $V^{\pi}(s_t) = \sum_{i=1}^{infinito} \gamma^{i-1} r_{t+i}$ (for a discounting case)
- The action value function Q^{Π}.

  And this can be calculated: $Q^{\pi}(S_t, a) = \sum_{i=1}^{infinito} \gamma^{i-1} r_{t+i}$ *where* $V^{\pi}(S_t) = Q^{\pi}(S_t, \pi(S_t))$

In a non-deterministic world, comming from a state S_t the agent usually has more than one posible destination state at t+1, each of those with a probability on ending there (given by the model T). So the calculation formula for V (S_t) and Q (S_t,a) must be changed adding the probabilistic term of the likelyhood of finishing on that state, it is:

$$V^{\Pi}(S_t) = \sum_{i=1}^{infinito} \gamma^{i-1} r_{t+i} = r_{t+1} + \gamma \cdot V^{\Pi}(S_{t+1}) \qquad \textit{for the } \textbf{\textit{deterministic}} \textit{ case}$$

$$V^{\Pi}(S_t) = r_{t+1} + \gamma \sum_{\forall i} T(S_t, \Pi(S_t), S_i) \cdot V^{\Pi}(S_i) \qquad \textit{for the } \textbf{\textit{non}} - \textbf{\textit{deterministic}} \textit{ case}$$

In the case of a non deterministic world, we'll have to take a mean (in spanish, media) of the possible states to which the agent will go (because, been not deterministic we cannot say what is the next state he'll jump to).

Once the agent is in the S_t state it can be calculated the difference between the real $V^{\Pi}(S_t)$ and the one estimated. The error will be:

$$error_t = r_{t+1} + \gamma \sum_{\forall i} T(S_t, \Pi(S_t), S_i) \cdot \widehat{V^{\Pi}}(S_i) - \widehat{V^{\Pi}}(S_t)$$

Actually, what the agent will learn is the estimation of the state value function $\hat{V}^{\Pi}$, not the real V^{Π}. The error will indicate how much he has learnt the function V^{Π} (zero for a perfect learning).

The same can be applied to the *action value function*:

$$Q^{\Pi}(S_t, a) = r_{t+1} + \gamma \sum_{\forall i} T(S_t, \boldsymbol{a}, S_i) \cdot V^{\Pi}(S_i) \qquad \text{for the } \textbf{non-deterministic case}$$

3.2. Ordering policies

There exists an operator that will tell us when a policy is better than another one: a policy Π' is better than another one Π when the state value function for all the states following Π' is greater than the state value function following Π.

Given a determined problem, there will be a group of possible policies to follow. An order can be stablished to detemine which of those policies are the better (a policy Π is better than or equal to another policy Π' if its expected return is greater than or equal to that of Π' for all state). Those policies which its state value function is greater or equal to the state value function of the rest of policies, for every state S are called **optimal policies**. Usually, for a given problem there exist several optimal policies.
This is expressed like this:

$$\Pi \;\; is\, better\, than \;\; \Pi' \;\; if \quad V^{\Pi'}(S) \geqslant V^{\Pi}(S) \quad \forall S$$

Policies could be *deterministic* or *non deterministic* (and this is independent of the kind of problem that we are working on. We could have a deterministic policy for a non deterministic problem). In case of a non deterministic policy, it will be expressed as Π(a,s), and its value won't be the action to perform but the probability of performing action a been at state s. We will work only with deterministic policies.

All the <u>optimal</u> policies that apply to the same problem, have the same value functions (state and action). So we'll only have to find one optimal state value function and we'll have an optimal policy. We'll call * an optimal policy, and V* and Q* the value functions following the policy * (there is no need to especify the sup-index in the V and Q optimal functions, since any optimal policy (or ' or '' or ...) will all have those same value functions). Optimal value functions are calculated as following:

$$V*(S) = \max_{\pi} V^{\pi}(S)$$

$$Q*(S, a) = \max_{\pi} Q^{\pi}(S, a)$$

$$so, the\, corresponding\, optimal\, policies \quad \pi*(s) = arg\max_{a} Q*(s, a)$$

Ex:

Q*(S,a1) = 0.2
Q*(S,a2) = 0.5
Q*(S,a3) = 1.2
Q*(S,a4) = 0.34

$\Pi^*(s) = a3$

A policy is **greedy** with respect to a value function if it is optimal according to that value function for a one-step problem. The non greedy policies are stupid policies.
The way of obtaining greedy policies from values is using the following relations:

$$\Pi(S_i) = \underset{a \in A}{argmax}\left(\sum_{\forall j} T(S_i, a, S_j) V(S_j)\right) \qquad for\ the\ non-deterministic\ case$$

$$\Pi(S_i) = \underset{a \in A}{argmax}\, Q(S_i, a) \qquad for\ the\ deterministic\ case$$

And the relation between V and Q in greedy policies:

$$V^{\Pi}(S_t) = \max Q^{\Pi}(S_t, a)$$

4. Methods for finding policies

4.1. Solve the Bellman equations

One way of finding an optimal policy is to solve Bellman equations. However this solution is rarely used, because it relies on the assumption that we accurately know the dynamics of the environment and that we have enough computational resources to find the solution (solving the equations is akin to an exhaustive search, looking for all the possibilities, and this requires huge resources when the problem is minimally big). Also the assumption of a MPD is required.
This solution is almost never used. Instead dynamic programming is used.

4.2. Dynamic programming

It "*... refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment*" [1].
We will assume that the environment is a MDP, it is, its state and sets of actions are given by a set of transition probabilities and expected inmediate rewards.

The main idea behind dynamic programing is the use of value functions to organize the search for good policies.
DP will be used then to compute the value functions V* and Q*.

4.2.1. Policy iteration

This is a process in two steps: policy evaluation and greedification.

- We start computing the state-value function V for an arbitrary policy .
- We look for a situation where Q (S,A) > V (S) (this is policy evaluation).
- Then we change to do A in the state A (greedification).

We continue the iteration until we find that in two consecutive steps we do not obtain any improvement.
If we assume that our domain is Markovian, this process will lead us to an optimal policy.
The number of iterations required to find an optimal policy will be bellow S iterations (the number of states availables is the limit).

The algorithm is the following:

- Choose an arbitrary policy
- Repeat the following until no improvement (i.e., while ' is different of)
  - for each state compute the state value function V (s)

$$V^{\Pi}(s) = \sum_{s' \in S} (r(s') + \gamma V^{\Pi}(s')) T(s, \Pi(s), s')$$

– for each state improve the policy at each state '(s)

$$\Pi'(s) = \underset{a \in A}{argmax} \left(\sum_{s' \in S} (r(s') + \gamma V^{\Pi}(s')) T(s, a, s') \right)$$

– = '

This algorithm is slow if there are a lot of states. That is why is possible to modify the cycle of iteration, doing iteration for a fixed number of states instead of all the states (but then it arises the problem of how to decide when to leave the repeat loop, and also how to decide through which states to iterate. It must be assured that all the states will be selected. This can be solved by randomly selecting the states and until all the states have been selected).

4.2.2. Value iteration

In this case we take a single step to do the valuation and the maximisation (greedification) of the function.
It doesn't uses policies, it just uses values. While V is improved it repeats for each state the following formula:

$$V(s) = \underset{a \in A}{max} \sum_{s' \in S} (r(s') + \gamma V(s')) T(s, a, s')$$

After finishing the loop, it must calculate the policy from the V obtained in the loop.
Same considerations as in the previous algorithm can be applied here.

To apply those algorithms it is necessary previously to obtain a model of the world. This is difficult in most of the cases if not impossible where the world is changing: if the world changes, the policy must change too but it won't change in this case because the model remains forever the same. Moreover, those algorithms need lots of resources.
All these drawbacks are overcome by the **reinforcement learning** algorithms.

4.2.3. Asynchronous versions

(No data available)

4.2.4. Reinforcement Learning algorithms

These are algorithms that don't need a model of the world. The agent learns the model through running in the world (learning by doing). They don't perform updates for all the states, just the ones interested on (on the extreme case, only one state is updated, the current state).
RL algorithms don't need a model of the world and are able to react in front of changes in the environment. They also have the advantage that have good policies before learning the optimal one.

The most popular RL algorithm is known as **Temporal Difference learning** (TD-learning or TD-backup).
The phylosophy behind this algorithm is the following: since we don't have the values of the model (this is T, the one which models the world), we do an estimation of it, by using previous known values obtained in previous experiments. If we run the experiment infinite times, the result will be the same as in dynamic programming.
Learning is performed by updating the value function on an state S at an instant t, using the value functions of the state at the next instat t+1.

$$V(S_t) = (1 - alfa) V(S_t) + alfa (r_{t+1} + \gamma V(S_{t+1}))$$

where is one by the number of experiments done. When the number of experiments becomes big, that calculation converges to the esperanza of $V(S_t)$. This way of calculation allows to change the value of $V(S_t)$ every time the agent is on that state S_t, and that new value will be a combination of the present reward and all the previous ones. As a result the agent will increasingly learn his environment without having to use a model. If the agent goes through that state enough times, its value function will have no difference with that calculated by policy iteration (the agent would have acquired the model of the world).
Re-arrenging the terms of the previous formula, we can obtain:

$$V(S_t) = V(S_t) + alfa(r_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

been the last part of the expression the TD error

$$Error = r + \gamma V(S') - V(S)$$ ¿por qué? ¿Qué significa este error?

The same can be applied to calculate Q.

$$Q(S_t, a_t) = Q(S_t, a_t) + alfa(r_{t+1} + \max_a Q(S_{t+1}, a_{t+1}) - Q(S_t, a_t))$$ [4.1] ¿Esta esto bien?

Agents making use of TD algorithms are always learning, it is, the process of changing the values of the functions never ends. This allows the agent to adapt himself to any change produced in the world (i.e., he will change his internal model of the world at any opportunity)[1].

Special cases of TD algorithms:

(a) Q-learning.

This is a TD(0) algorithm.

It uses equation 4.1 to re-calculate Q(s,a) at every step that the agent changes state, and updates its policy for that state *s*.

The algorithm works as follows:

- the agent initialices Q (s,a) and Π(s) arbitrarily, and sets initial state *s*.
- then it enters into a never ending loop
    - uses the policy to determine the action to perform: a=Π(s)
    - executes the action *a*, gets the reinforcement *r* and perceives the new state *s'*
    - updates the Q function value at state *s* having done action *a*

$$Q(s, a) = Q(s, a) + alfa(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

    - since the value of Q at *s* has changed (Q(s,a)), it is possible that the current policy at state *s* (Π (s)) is not now optimal. So, the agent updates the policy taking into account the new value of Q (s,a)

$$\Pi(s) = \arg\max_{a \in A} Q(s, a)$$

    - the current state is now *s'* (s = s'). End of the loop. Repeat all.

This algorithm does not assures that the agent will find an optimal policy, because some of the states will not be visited during the calculation of their value functions. To avoid this problem the agent uses use **exploratory actions**: this means that, sometimes, even when the policy says that the action to take is *a*, the agent will take an action by chance *a'*.

If the agent doesn't follow always the rules he lets room to explore different possibilities[2] than the ones rigidly established by the policy. Those exploratory actions can lead to worse behaviors, but also can allow the agent to find better policies than the present one.

An value is defined to determine the **frequency of the exploratory actions** (called - **greedy**). By definition, the -greedy method assigns the same probability to all exploratory states. Sometimes this is a bad solution (in cases with two states with similar values). Then a **softmax** method is defined where a probability is defined for every exploratory states following the following formula:

$$P_s(a) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_{b \in A} e^{\frac{Q(s,b)}{T}}}$$

1 Author's opinion: A drawback is that he will not be able to maintain knowledge from previous states of the world, since the agent is too well-disposed to change his behavior (I would say that he is very **influenciable** by the current *fashion*).

2 A wise person must break the rules from time to time, but he must know when to do it.

A way to avoid exploratory actions would be initialising Q values optimistically (no lo entiendo!!). This is useful on static worlds, but not on worlds that change.

Taking exploratory action into account, the final Q-learning algorithm would be (the only difference with the previous version is that we introduce the exploratory action):
- the agent initialices Q (s,a) and Π(s) arbitrarily, and sets initial state s.
- then it enters into a never ending loop
    - selects the action *a* to perform: he either uses the policy to determine the action (a=Π(s)) or he takes an exploratory action
    - executes the action *a*, gets the reinforcement *r* and perceives the new state *s'*
    - updates the Q function value at state *s* having done action *a*

$$Q(s,a)=Q(s,a)+alfa(r+\gamma \max_{a'} Q(s',a')-Q(s,a))$$

    - since the value of Q at *s* has changed (Q(s,a)), it is possible that the current policy at state *s* (Π(s)) is not now optimal. So, the agent updates the policy taking into account the new value of Q (s,a)

$$\Pi(s)=\underset{a\in A}{argmax}\, Q(s,a)$$

    - the current state is now *s'* (s = s'). End of the loop. Repeat all.

When the agent follows the action determined by the policy, it is said that the agent is doing an **exploitation**. When the agent takes an exploratory action, it is said that he is doing an **exploration**.

Actually, α depends on S, since α is diferent for every state *s* since the states will be visited different times (in that case Q and V values would be exact arithmetic average of the experiences). In practice, α takes a constant value; this is more efficient, but it has other advantages.
A constant α value would mean that experiments in the past are loosing importance with an exponential order in the final calculation of the mean. This is good for a world that changes, because as the policy is changing with the change of the world, so the values must change to be up to date with the new world. So, a constant value for α gives more importance to recent values (experiences).
The Q-learning algorithm will converge if three conditions are met (see slides for more info). In practice, the algorithm converges even if some of the conditions are not met.

(b) Sarsa Backup algorithm

This is another TD(0) algorithm.
The difference with the Q-learning algorithm is that the actualization of Q is done using the action taken by the agent, not the action that the policy says that you he will take. The difference is not very high except when the system is doing exploration. In this case sarsa will compute with the real values of the actions taken, but Q-learning will use the value predicted by the policy.
The update of the Q value function is done in the following way:

$$Q(s_t,a_t)\leftarrow Q(s_t,a_t)+alfa[\underbrace{r_{t+1}+\gamma Q(s_{t+1},a_{t+1})-Q(s_t,a}_{TD\ Error}$$

The important issue here is that the update of Q (s_t,a_t) is not performed when the agent is in state s_t, but it is done when the agent leaves s_t state and has already done action at, so he actually knows which is the s_{t+1} state.

The sarsa algorithm:

- Initialize Q(s,a) and Π(s) randomly
- Set the initial state of the agent
- Select an action a, depending on the action-selection procedure (a greedy action or an exploratory one)

- Execute the action selected, get the reinforcement r and perceive the new state s'
-Then repeat the following loop forever
  - select new action a' depending on the action-selection procedure and the current state s'
  - Q (s,a) = Q(s,a) + (r + Q(s',a') - Q(s,a))
  - $\Pi(s) = \underset{a \in A}{argmax}\, Q(s,a)$
  - execute action a', get reinforcement r' and perceive new state s''
  - r = r' ; s = s' ; a = a' ; s' = s''

Sarsa is an **on-line** algorithm, it is, values are calculated from the **real** policy the agent is applying. Q-learning is an **off-line** algorithm because it learns a value function from a policy that is not the real policy applied

Q-learning and sarsa are the most widely implemented algorithms in practical systems.

(c) Monte Carlo

This is a TD(1) algorithm. It's only applicable to experiments with an end, limited by time or by finishing the task.
The value functions are only updated when the agent finishes the task based on the final total reward obtained.
The reward for a state s_t is the addition of all the rewards obtained on every step taken after this state s_t.. Nevertheless, the final value functions are calculated after a series of experiments, by taking a mean of the rewards obtained on every experiment realised .
The problem with monte carlo is that the agent only learns when finishes the task, but Q-learning learns even during the proces of finishing the task.

(d) N-steps estimators

Making a generalisation from what has been shown up to now about TD algorithms, a complete range of estimators can be created. Let's see.
It is known that

$$Q(s_t,a) = r_t + \gamma V(s_{t+1})$$

But it could have said also that

$$Q_2(s_t,a) = r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2})$$
$$Q_3(s_t,a) = r_t + \gamma r_{t+1} + \gamma r_{t+2}^2 + \gamma^3 V(s_{t+3})$$
$$\dots$$
$$Q_n(s_t,a) = \sum \gamma^{i-1} r_{t+i} + \gamma^n V(s_{t+n})$$
$$\dots$$
$$Monte\,Carlo\,estimator$$

Any of the previous sentences can be used to calculate the error in terms of the estimation of the 'n' next states. This is called the **n-steps estimator** and it leads to the definition of a new TD algorithm called **TD-λ.**
For any 'n' selected, the update of the value function for one state s_t is done in the following way:

$$Q_n(s_t,a_t) = Q_n(s_t,a_t) + alfa[\sum_{i=1}^{n} \gamma^{i-1} r_{t+1} + \gamma^n V(s_{t+n}) - Q^n(s_t,a_t)]$$

(e) TD-

This method is a unification of the learning algorithms. What it does is to calculate the geometric average of several n-step experiments, it is, the agent runs an experiment, then he calculates the value of $V(s_t)$ using a 1-step estimator, also calculates the $V(s_t)$ using a 2-steps estimator, and also using a 3-step estimator,..., and using a n-step estimator. Then, to calculate final value for $V(s_t)$, the agent calculates a

geometrical average[3] of all those values with parameter λ (the way the mean is taken is controlled by parameter).
So, the new definition of the estimation of the value function is a geometrical average with parameter λ.

$$V^{\lambda}(s_t)=(1-\lambda)\sum \lambda^{n-1}V^{n}(s_t)$$

with λ = 0 we obtain simple TD (Q-learning)
with λ = 1 we obtain Monte Carlo

The backup of the value of one state V_t is calculated then as follows:

$$V(s_t) = V(s_t)+ \alpha(V^{\lambda}(s_t) - V(s_t))$$

The problem here is that in order to perform the back-up of the value it will be necessary to end the current experiment, because only then will the agent have all the values required to do the calculation. This is a real problem since it is required to calculate the values during the execution of the experiment not after the experiment has finished (there could be cases where the experiment never ends). Fortunately that calculation is equivalent to a calculation using the **traces** of past events. (Por qué?. No entiendo porque es equivalente...).
The previous way of calculating the back-up is called the **forward view** of the TD() algorithm. There exists a **backward view** of the algorithm that allows the practical calculation of the back-up, and it is achieved by using **elegibility traces**.
A trace is an additional memory variable associated with each state, and for a state s at time t it is denoted $e_t(s)$. Traces record which states have **recently** been visited. The definition of a trace is as follows:

$$e_t(s)=\begin{pmatrix} \gamma\lambda e_{t-1}(s) & if \quad s\neq s_t \\ \gamma\lambda e_{t-1}(s)+1 & if \quad s=s_t \end{pmatrix}$$

The term *recently* is defined in terms of $\gamma\lambda$, where γ is the discount rate and λ acts like the trace-decay parameter.

No acabo de ver como funciona la versión backwards del algoritmo y qué papel tiene en él la traza.

The TD(λ) algorithm is as follows:

- Initialize V(s) randomdly and e(s)=0 for all s
- Then repeat the following loop forever
  - Take a first state s
  - Then repeat the following loop until end of the experiment
    - Select the action to take a given V and s (with exploratory strategy)
    - Take action a, receive reward r and perceive next state s'
    - credit = (1-λ)(r+γ(V(s')-V(s)))
    - e(s) = e(s) + 1
    - For all the states s'' which trace <> 0
      - V(s'')=V(s'') + α · credit · e (s'')
      - e(s'') = $\gamma\lambda$e(s'')
    - s=s'

There exist a variation of the definition of trace consisting on replacing traces instead of accumulating them, it is:

$$e_t(s)=\begin{pmatrix} \gamma\lambda e_{t-1}(s) & if \quad s\neq s_t \\ 1 & if \quad s=s_t \end{pmatrix}$$

This has the advantadge that *e* has a coated value.

3 Taking a geometrical average instead of an arithmetic one is a practical issue, but also is similar to the natural decay that a real neuron suffers on activation.

1. Problems with Reinforcement Learning algorithms

The typical problem with RL algorithms arises from the requirement of having an MDP environment (it is, only the information about the current state is necessary to learn intelligently). Reallity shows that sometimes that requirement is not met putting the agent in a difficult situation and making him to make mistakes..
A way to avoid this problem while staying in a MDP system is using a mechanism known as POMDPs (Partially Observable MDPs). This method assigns probabilities of been in one state given an observation (P(O,S)). That probability indicates the believe of the agent of been in state S given the observation O. The agent must know *a priori* all the possible states where the agent could be. The method works as follows:
before an observation is made, the agent believes that he could be in any of the possible states. After doing an observation, the agent reduces that believe (since just a few states will have correspondence with the observable). Then the agent makes a move, reducing in this way again the likelihood of been in some states. This process is repeated until only one state retains probability 1, so the agent knows he is on that state. From there on, the agent knows where he is and can then apply any policy.
A drawback of this method is the necessity of lnowledge of the possible states and their P(O,S) values.

A possible application of this method is for a robot that wants to know where he is now having just a partial knowledge of the place and without hte necessity of creating and maintaining a map of the place.

2. References:

[1] *Reinforcement learning: an introduction* by Richard Sutton and Andrew Barto Ed. The MIT Press
It contains some code and slides on its web page (http://www-anw.cs.umass.edu/~rich/book/the-book.html)

[2] *Neurodynamic programming* by Bertsekas and Tsilikis. Using ANNs for the representation of knowledge