## Reinforcement learning

In many domains it is difficult to give a precise evaluation function, which would allow an agent to evaluate the effectiveness or correctness of her actions, except after she has reached a terminal state. In the terminal state the agent by default obtains an objective evaluation of her actions which is termed a **reward**, or a **reinforcement**. It is convenient to state the agent's task so that it would have to act in such a way to maximize this reward or reinforcement. This is called **reinforcement learning**.

In a general case the agent may not have full information about her environment, as well as a precise, or any, description of her actions. The situation of this agent is similar to one of the possible statement of a full task of artificial intelligence — the agent is placed in an environment she does not know, and cannot act in it. She has to learn to act in this environment effectively, to maximize some criterion, available as reinforcements.

We shall consider a probabilistic model of the agent's action outcomes. In fact, we shall assume that we are dealing with an unknown Markov decision problem (MDP).

## Passive reinforcement learning

Let us first consider **passive reinforcement learning**, where we assume that the agent's policy $\pi(s)$ is fixed. Agent is therefore bound to do what the policy dictates, although the outcomes of her actions are probabilistic. The agent may watch what is happening, so she knows what states she is reaching and what rewards she gets there.

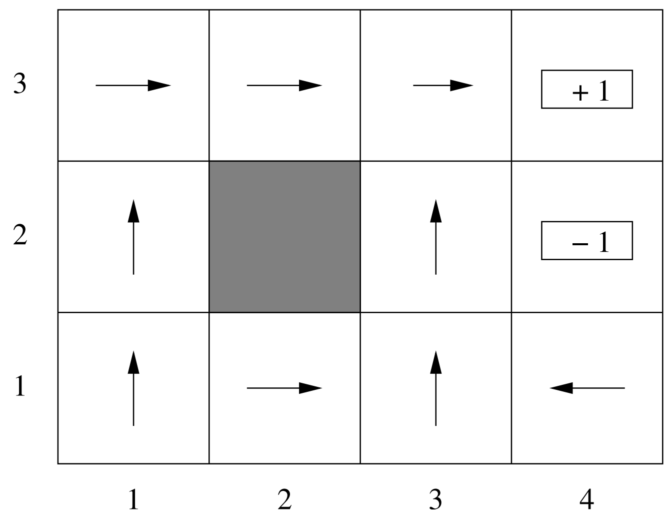
The agent's jobs is to learn the utilities of the states $U^\pi(s)$, computed according to the equation:

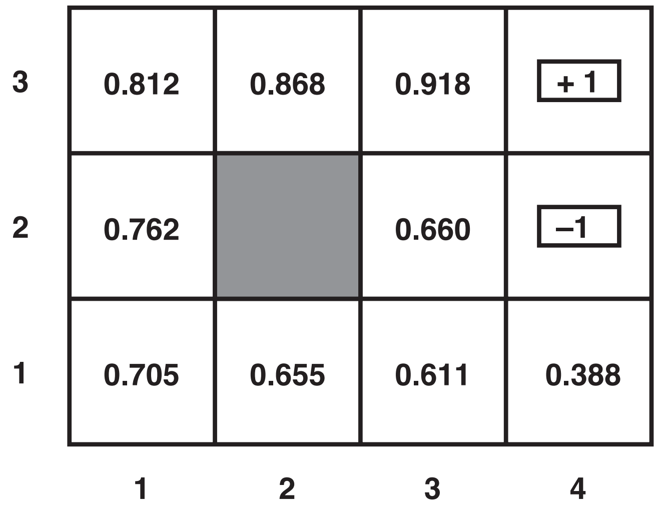
$$U^\pi(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right]$$

For the 4x3 example environment used here for illustration we will assume $\gamma = 1$.

# Trials

Recall the $4 \times 3$ environment we studied earlier:





The agent executes the **trials** where she performs actions according to the policy, until reaching a terminal state, and receives percepts indicating both the current state and the reinforcement. Example trials:

$(1,1)_{-0.04} \rightsquigarrow (1,2)_{-0.04} \rightsquigarrow (1,3)_{-0.04} \rightsquigarrow (1,2)_{-0.04} \rightsquigarrow (1,3)_{-0.04} \rightsquigarrow (2,3)_{-0.04} \rightsquigarrow (3,3)_{-0.04} \rightsquigarrow (4,3)_{+1}$

$(1,1)_{-0.04} \rightsquigarrow (1,2)_{-0.04} \rightsquigarrow (1,3)_{-0.04} \rightsquigarrow (2,3)_{-0.04} \rightsquigarrow (3,3)_{-0.04} \rightsquigarrow (3,2)_{-0.04} \rightsquigarrow (3,3)_{-0.04} \rightsquigarrow (4,3)_{+1}$

$(1,1)_{-0.04} \rightsquigarrow (2,1)_{-0.04} \rightsquigarrow (3,1)_{-0.04} \rightsquigarrow (3,2)_{-0.04} \rightsquigarrow (4,2)_{-1}$

## Direct utility determination

The objective of the agent is to compute the state utilities $U^{\pi}(s)$ generated by the current policy $\pi(s)$. The state utilities are defined as expected values of the sums of the (discounted) reinforcements received by the agent, who started from a given state, and is acting according to her policy:

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t)\right]$$

The agent may approximate the above quantity using the trials by computing the **reward-to-go** for each state visited along the way. At the end of each trial the agent takes the reward obtained in that state to be its utility, and then, backtracking along the trial, computes the reward-to-go for subsequent states, by summing up the rewards obtained in the last section of the trial.[1]

For example, for the trial:

$(1,1)_{-0.04} \leadsto (1,2)_{-0.04} \leadsto (1,3)_{-0.04} \leadsto (1,2)_{-0.04} \leadsto (1,3)_{-0.04} \leadsto (2,3)_{-0.04} \leadsto (3,3)_{-0.04} \leadsto (4,3)_{+1}$

we get $R_{tg}(4,3) = 1, R_{tg}(3,3) = 0.96, R_{tg}(2,3) = 0.92, R_{tg}(1,3) = 0.88, R_{tg}(1,2) = 0.84, R_{tg}(1,3) = 0.80, R_{tg}(1,2) = 0.76, R_{tg}(1,1) = 0.72$

[1] In case of discounting coefficient $\gamma \neq 1.0$ the rewards should be properly discounted.

By averaging over a large number of samples she can determine the subsequent approximations of the expected value of state utilities, which converge in the infinity to the real expected values. This way the reinforcement learning task is reduced to a simple inductive learning.

This approach works, but is not very efficient, since it requires a large number of trials. The problem is, that the algorithm, by using simple averaging, ignores important information contained in the trials, namely, that state utilities in neighboring states are related.

$(1,1)_{-0.04} \rightsquigarrow (1,2)_{-0.04} \rightsquigarrow (1,3)_{-0.04} \rightsquigarrow (1,2)_{-0.04} \rightsquigarrow (1,3)_{-0.04} \rightsquigarrow (2,3)_{-0.04} \rightsquigarrow (3,3)_{-0.04} \rightsquigarrow (4,3)_{+1}$

$(1,1)_{-0.04} \rightsquigarrow (1,2)_{-0.04} \rightsquigarrow (1,3)_{-0.04} \rightsquigarrow (2,3)_{-0.04} \rightsquigarrow (3,3)_{-0.04} \rightsquigarrow (3,2)_{-0.04} \rightsquigarrow (3,3)_{-0.04} \rightsquigarrow (4,3)_{+1}$

$(1,1)_{-0.04} \rightsquigarrow (2,1)_{-0.04} \rightsquigarrow (3,1)_{-0.04} \rightsquigarrow (3,2)_{-0.04} \rightsquigarrow (4,2)_{-1}$

For example, in the second trial of the previous example, the algorithm evaluates the utility of state (3,2) as the reward-to-go only from this trial, but ignores the fact, that the successor state in this state is (3,3), which has an already known, significantly higher utility. The Bellman equation relates the utilities of successor states, but this approach cannot take advantage of this.

## Adaptive dynamic programming

The **adaptive dynamic programming** (ADP) is a process similar to the dynamic programming, combined with learning the model of the environment, ie. the state transition probability distribution and the reward function. It works by counting the transitions from the state-action pairs to the next states. The trials provide the training data of such transitions. The agent can compute the probability of the transitions as frequencies of their occurrences in the trials.

For example, in the presented trials, the action $\boxed{\rightarrow}$ (Right) was executed three times in the state (1,3). Two of these times the successor state was (2,3), so the agent should compute $P((2,3)|(1,3), \textit{Right}) = \frac{2}{3}$.

$(1,1)_{-0.04} \leadsto (1,2)_{-0.04} \leadsto (1,3)_{-0.04} \leadsto (1,2)_{-0.04} \leadsto (1,3)_{-0.04} \leadsto (2,3)_{-0.04} \leadsto (3,3)_{-0.04} \leadsto (4,3)_{+1}$

$(1,1)_{-0.04} \leadsto (1,2)_{-0.04} \leadsto (1,3)_{-0.04} \leadsto (2,3)_{-0.04} \leadsto (3,3)_{-0.04} \leadsto (3,2)_{-0.04} \leadsto (3,3)_{-0.04} \leadsto (4,3)_{+1}$

$(1,1)_{-0.04} \leadsto (2,1)_{-0.04} \leadsto (3,1)_{-0.04} \leadsto (3,2)_{-0.04} \leadsto (4,2)_{-1}$

After executing every single action the agent updates the state utilities by solving the (simplified) Bellman equation using one of the appropriate methods. The equation is simplified because we only know the distribution of the effects of actions belonging to the policy, and we cannot compute the best action for each state. Since we are computing the $U^{\pi}$ we take just these actions.

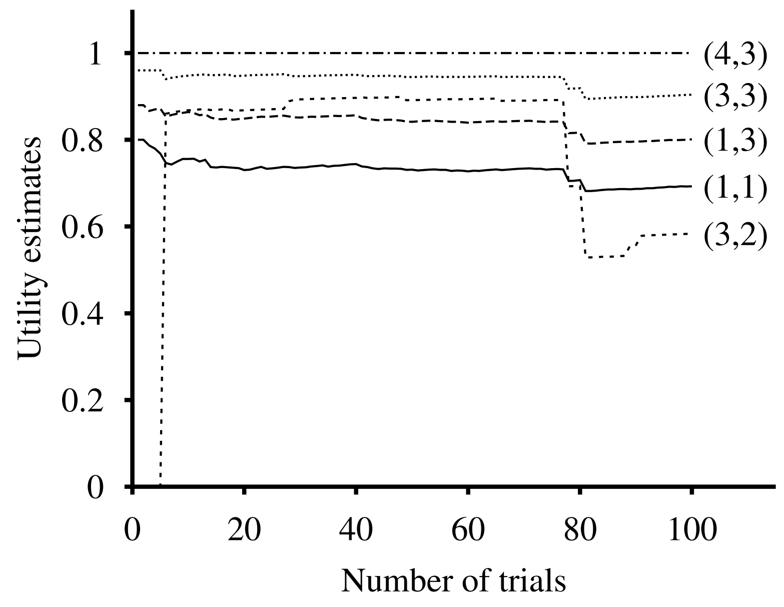
## Adaptive dynamic programming — the algorithm

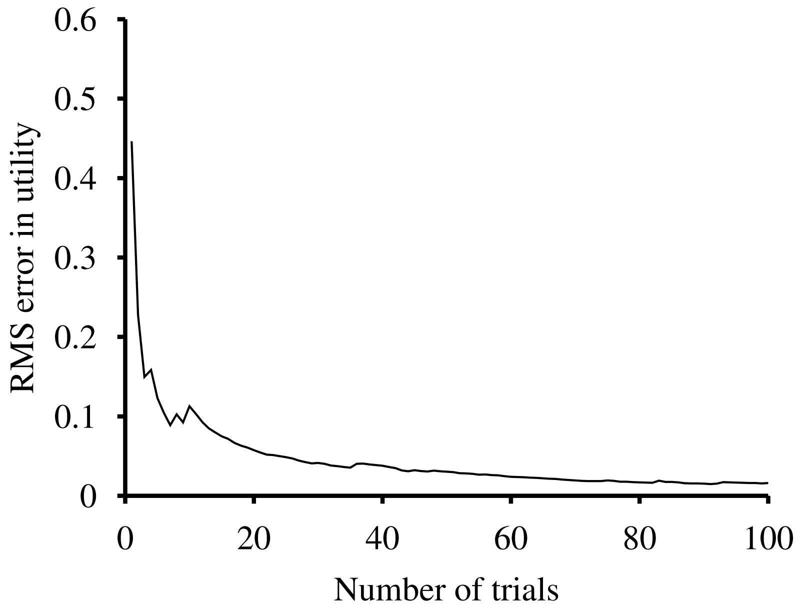
```
function PASSIVE-ADP-AGENT($percept$) returns an action
  inputs: $percept$, a percept indicating the current state $s'$ and reward signal $r'$
  persistent: $\pi$, a fixed policy
              $mdp$, an MDP with model $P$, rewards $R$, discount $\gamma$
              $U$, a table of utilities, initially empty
              $N_{sa}$, a table of frequencies for state–action pairs, initially zero
              $N_{s'|sa}$, a table of outcome frequencies given state–action pairs, initially zero
              $s$, $a$, the previous state and action, initially null

  if $s'$ is new then $U[s'] \leftarrow r'$; $R[s'] \leftarrow r'$
  if $s$ is not null then
      increment $N_{sa}[s, a]$ and $N_{s'|sa}[s', s, a]$
      for each $t$ such that $N_{s'|sa}[t, s, a]$ is nonzero do
          $P(t \mid s, a) \leftarrow N_{s'|sa}[t, s, a] \,/\, N_{sa}[s, a]$
  $U \leftarrow$ POLICY-EVALUATION($\pi$, $U$, $mdp$)
  if $s'$.TERMINAL? then $s, a \leftarrow$ null else $s, a \leftarrow s', \pi[s']$
  return $a$
```

## Adaptive dynamic programming — efficiency

The ADP algorithm updates the utility values as best as it is possible, and in this respect it is a standard that the other algorithms can be compared to. The policy computation procedure, which solves a system of linear equations, can be intractable for problems with large state spaces (eg. for the backgammon game we get $10^{50}$ equations with $10^{50}$ unknowns).





The above charts show the convergence for an example learning experiment for the $4 \times 3$ environment. In this experiment the first trial ending in the "bad" terminal state first occurs around the 78th trial, which causes large updates to some utility values.

## Temporal difference learning

Instead of solving the full equation system for each trial, it is possible to update the utilities using the currently observed reinforcements. Such algorithm is called the **temporal difference** (TD) method:

$$U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma U^\pi(s') - U^\pi(s))$$

In this case the single utility value is updated from a single observed state transition, instead of the expected value of all transitions. This is why we take this correction — the difference between the utility of the move and the utility of the state — reduced by a factor $\alpha < 1$. This introduces small corrections after each move. The correction converges to zero when the state utility becomes equal to the discounted utility of a move.

Note that this method does not require having a model of the environment $P(s'|s,a)$, nor does it compute one.

## Temporal difference learning — the algorithm

```
function PASSIVE-TD-AGENT(percept) returns an action
    inputs: percept, a percept indicating the current state $s'$ and reward signal $r'$
    persistent: $\pi$, a fixed policy
                $U$, a table of utilities, initially empty
                $N_s$, a table of frequencies for states, initially zero
                $s, a, r$, the previous state, action, and reward, initially null

    if $s'$ is new then $U[s'] \leftarrow r'$
    if $s$ is not null then
        increment $N_s[s]$
        $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma U[s'] - U[s])$
    if $s'$.TERMINAL? then $s, a, r \leftarrow$ null else $s, a, r \leftarrow s', \pi[s'], r'$
    return $a$
```

## Convergence of the temporal difference method

There is a close relationship and similarity between the ADP and TD algorithms. While the latter makes only local updates in utility values, their average values converge to the same values as for the ADP algorithm.

In case of learning with many transition examples, the frequencies of state occurrences agrees with their probability distributions, and it can be proved that the utilities converge to the correct values.

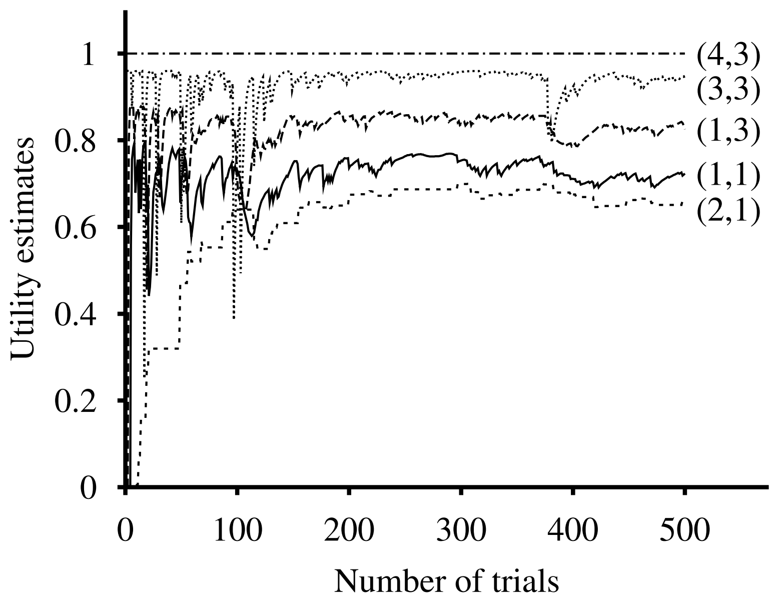
For this to occur the learning parameter $\alpha$ should decrease with the number of processed trials. More precisely, the subsequent values of this parameter should satisfy the requirements:

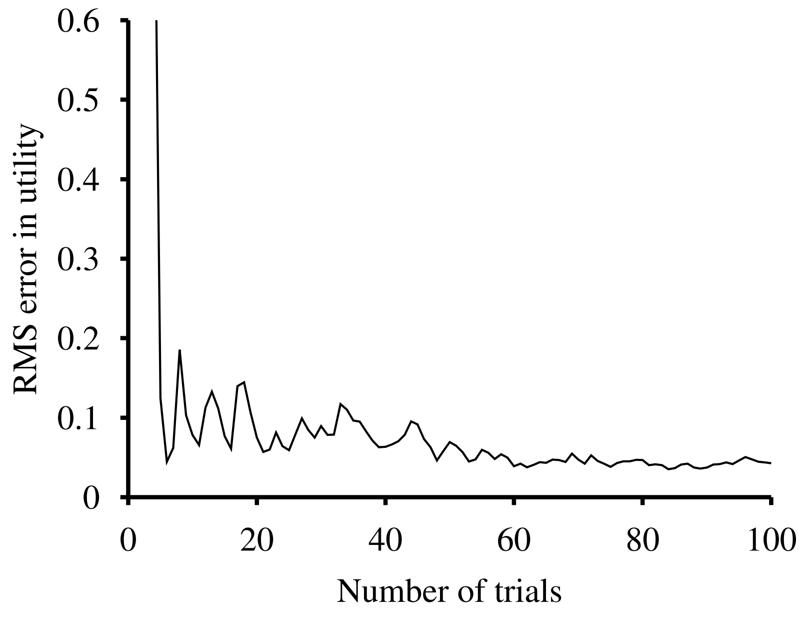
$$\sum_{n=1}^{\infty} \alpha(n) = \infty$$

and also:

$$\sum_{n=1}^{\infty} \alpha^2(n) < \infty$$

The convergence of another example learning experiment for the $4 \times 3$ environment:





## Active reinforcement learning

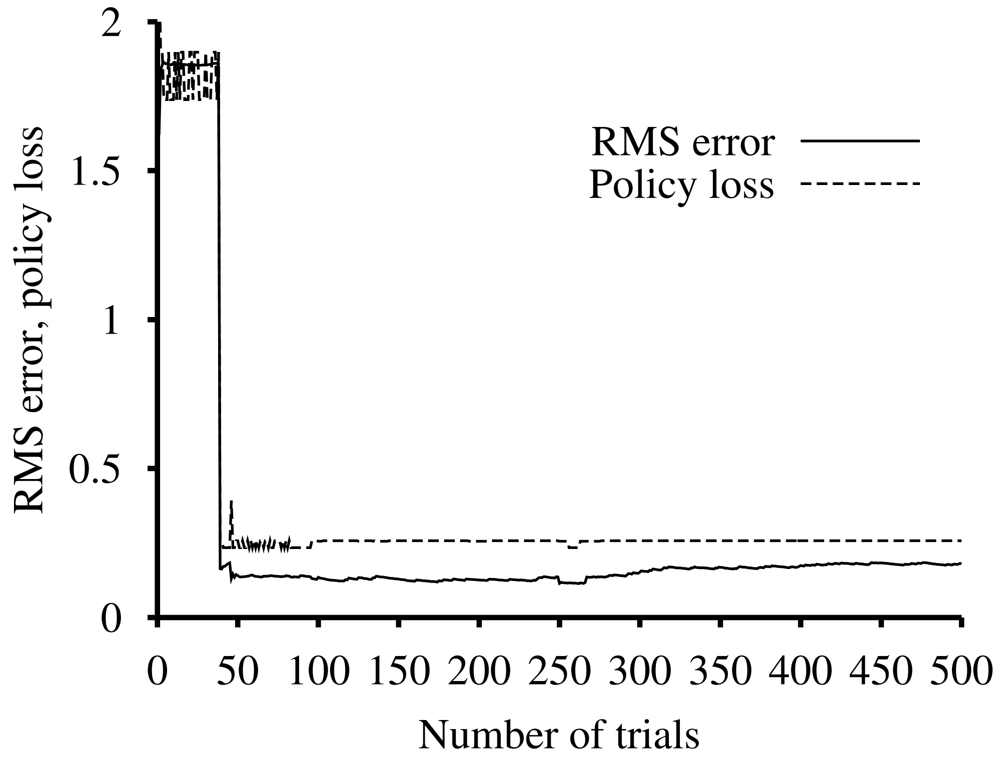
What should do an agent, which does not have a policy, or who would like to find an optimal one?

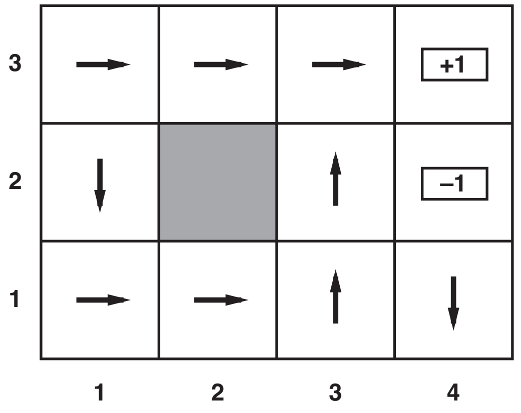
First she should compute the complete transition model for all the actions. The ADP algorithm for a passive agent can be used for that. After that, the optimal policy, satisfying the Bellman equation, can be determined like for a regular Markov decision process:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)U(s')$$

The agent can use the value iteration or the policy iteration algorithm. Then, having determined the optimal policy she could simply execute this policy.

But should she?





The chart on the left shows the result of learning for a sample experiment. The agent found a direct path to the [+1] solution in the 39th trial, but this was the worse path, along the states: (2,1), (3,1), (3,2), (3,3). This has, however, determined the agent's computed optimal policy, on the right. It turns out to be typical in this environment, that the agent only rarely arrives at the optimal policy preferring the "upper" path: (1,2), (1,3), (2,3), (3,3).

## Exploration

Unfortunately, if the agent does not learn the correct environment model from the initial series of trials, and computes the optimal policy for the model it learned, then any subsequent trials will be generated according to the policy possibly suboptimal for the given environment.

This is the compromise between the **exploitation** of the knowledge already possessed, and the **exploration** of the environment. The agent should not too early accept the learned environment model, and the policy computed for it. She should try different possibilities.

Moreover, she should try multiple times all the actions in all the states, if she wants to avoid the possible situation, that an unlucky series of trials prevents her from discovering some particularly advantageous choices of actions. However, eventually she needs to start acting according to the optimal policy, to tune it to its specific patterns.

## The exploration policy

In order to combined the effective world exploration with the exploitation of the possessed knowledge, the agent must have some **exploration policy**. It is necessary to ensure that the agent would be able to learn all her available actions to the degree permitting her to compute the globally optimal policy.

A simple exploration policy would be to execute some random actions in all states some fraction of the time, and follow the optimal policy the rest of the time.

This approach works, but is slow to converge. It would be better to prefer the exploration of relatively unexplored state-action pairs, while at the same time avoiding the pairs already better known and believed to be of low utility.

## The exploration function

A reasonable exploration policy can be achieved by introducing the optimistic approximation of utilities $U^+(s)$:

$$U^+(s) \leftarrow R(s) + \gamma \max_a f\left(\sum_{s'} P(s'|s,a)U^+(s'), N(a,s)\right)$$

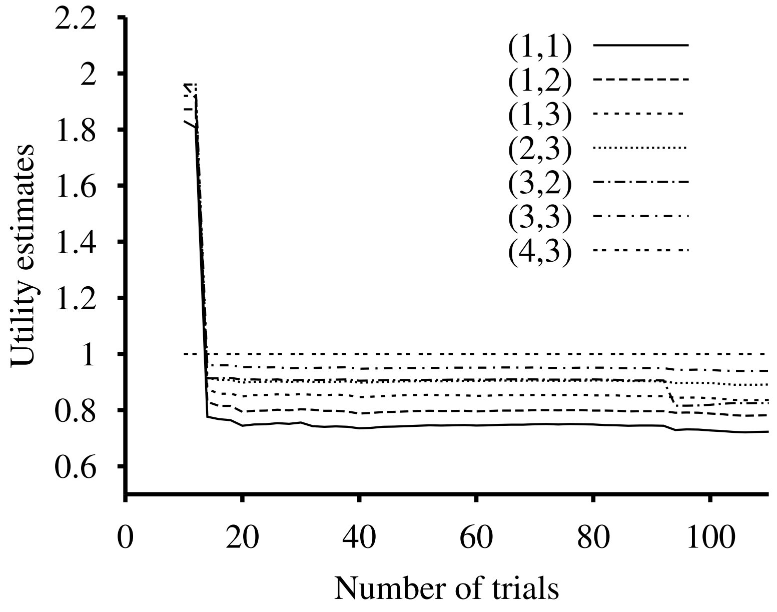
where $N(a,s)$ is the number of previous choices of action $a$ in state $s$, and $f(u,n)$ is the **exploration function**, trading off the greed (large values of $u$) against curiosity (small values of $n$).

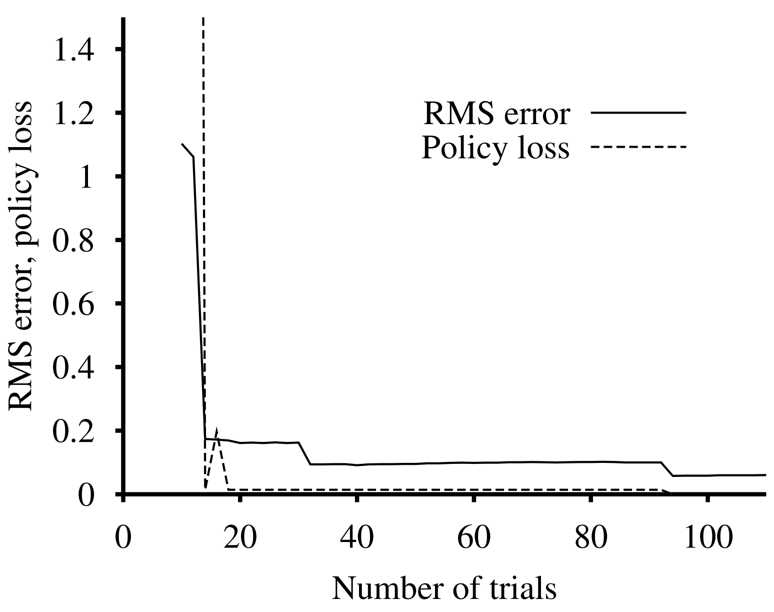
Obviously the $f$ function should be increasing with respect to $u$ and decreasing with respect to $n$. A simple definition of $f$ can be:

$$f(u,n) = \begin{cases} R^+ & \text{if } \; n < N_e \\ u & \text{otherwise} \end{cases}$$

where the $R^+$ denotes the optimistic estimate of the best reward possible to obtain in any state, and the $N_e$ is the minimal number of times that the agent will want to try each state-action pair.

The fact, that the update formula for $U^+$ in the right hand side also uses $U^+$ is important. Since the states and actions surrounding the starting state will be executed multiple times, if the agent used non-optimistic utilities for updating, she might get discouraged from such states, and start to avoid “setting out”. Using $U^+$ instead means, that the optimistic values generated around the newly explored regions will be propagated back, and the actions leading to unknown regions will be favored.





The chart on the left shows the results of learning with exploration. The policy close to optimal was reached after 18 trials. Note that the utility values converge more slowly (RMS error) than the optimal policy is determined (policy loss denotes the maximum loss from executing a suboptimal policy).

## Active temporal difference learning

The method of temporal differences can also be applied to active learning. The agent might not have a fixed policy, and still calculate the state utilities using the same update formula as in the passive case:

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s))$$

Given the computed utilities the agent can determine the optimal actions for each state using the utilities of successor states. It can be proved, that the active TD agent will eventually arrive at the same resulting utility values as the active ADP agent.

## Q-learning

An alternative to temporal difference learning is the Q-learning method, which learns an action-value representation in the form of the function $Q(a, s)$, which expresses the value of the action $a$ in state $s$, and is related to the utilities with the formula:

$$U(s) = \max_a Q(a, s)$$

The goal values of $Q$ satisfy the equation:

$$Q(a, s) = R(S) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(a', s')$$

The above formula can be utilized in the iterative process for updating the value of $Q$. However, this requires to simultaneously learn the $Q$ values, and the model, as the probability distribution $P$, which is present in the formula.

## Q-learning — updating by temporal differences

It is also possible to apply local updating of the $Q$ function, which is a variant of the temporal difference learning. It is given by the following updating formula, computed whenever action $a$ is executed in state $s$ leading to the result state $s'$:

$$Q(a,s) \leftarrow Q(a,s) + \alpha(R(s) + \gamma \max_{a'} Q(a',s') - Q(a,s))$$

Q-learning with temporal difference updating converges to the result much slower than the ADP algorithm since it does not enforce computing the full consistency of the model, which it does not have.

## The full Q-learning algorithm with exploration

```
function Q-LEARNING-AGENT(percept) returns an action
    inputs: percept, a percept indicating the current state $s'$ and reward signal $r'$
    persistent: $Q$, a table of action values indexed by state and action, initially zero
                $N_{sa}$, a table of frequencies for state–action pairs, initially zero
                $s, a, r$, the previous state, action, and reward, initially null

    if TERMINAL?(s) then $Q[s, None] \leftarrow r'$
    if $s$ is not null then
        increment $N_{sa}[s, a]$
        $Q[s,a] \leftarrow Q[s,a] + \alpha(N_{sa}[s,a])(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
    $s, a, r \leftarrow s', \text{argmax}_{a'} f(Q[s',a'], N_{sa}[s',a']), r'$
    return $a$
```

In general an active Q-learning agent requires exploration, just as it is in the case with the ADP method. That is why the algorithm uses the exploration function $f$ and the action frequency table $N$. With a simpler exploration function (eg. executing some fraction of random moves) the $N$ table might not be necessary.

## SARSA — *State-Action-Reward-State-Action*

There exists a variant of the Q-learning algorithm with a temporal difference update method called SARSA (*State-Action-Reward-State-Action*):

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma Q(s',a') - Q(s,a))$$

SARSA updates take into account five elements: $s, a, r, s', a'$. While the Q-learning algorithm adjustments are based on the <u>best</u> action selected for the state reached by the action $a$, SARSA considers which action was in fact selected. Therefore, eg. for a greedy agent executing only exploitation, these two approaches would give the same result.

However, in case of exploration the difference is significant. Q-learning is an **off-policy** learning algorithm, computing the best possible Q values, regardless of where the current policy takes us. On the other hand, SARSA is an **on-policy** algorithm, which is more appropriate for the agent acting according to the policy.

Q-learning is a more flexible algorithm, as it permits the agent to learn the optimal behavior even if she currently executes a policy different from the learned patterns. On the other hand, SARSA is more realistic, since, for example, if the agent was unable to fully control her policy, then it would be better for her to learn the patterns corresponding to what will in fact happen with her, than to learn the best possible patterns.

Both Q-learning and SARSA are capable of learning the optimal policy for the 4x3 example environment, albeit more slowly than the ADP (in terms of the number of iterations). This is because the local updates they make do not enforce the full consistency of the Q function.

Comparing these two methods we might take a broader look and ask, whether it is better to learn the model of the environment and the utility function, or to directly determine the mapping from states to actions, without paying attention to the environment model.

This is one of the fundamental questions of how the artificial intelligence should be constructed. For many years of its initial development, the paradigm of the **knowledge-based** systems dominated, postulating to build declarative models. The fact that recently model-free approaches such as the Q-learning emerge suggests, that perhaps this was not necessary. However, the model-based methods are still better for some complex tasks, so this issue remains open.

## Generalization in reinforcement learning

The above reinforcement learning algorithms assume an explicit representation of the $U(s)$ or $Q(s)$ function, such as, eg. table representation. This may be practical only up to some size of the problem.

For example, for problems with a very large number of states (eg. $\gg 10^{20}$ for games such as chess or backgammon), is is hard to envision executing a sufficient number of trials to visit each state frequently enough. It is necessary to use some generalization method, which would permit to determine an effective policy based on a small part of the state space explored.

## Function approximation

One of such methods is the **function approximation**, based on the representation of the function under examination (eg. $U$) in some nontabular form, like a finite formula. Similarly as was the case with the heuristic functions, some linear combination of some features (also called the state attributes) can be used:

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + ... + \theta_n f_n(s)$$

The reinforcement learning algorithm would learn the vector of coefficients $\theta =< \theta_1, \theta_2, ..., \theta_n >$ so that the evaluation function $\hat{U}_\theta$ would closely enough approximate the state utility function.

This approach is called a function approximation because there is no proof that the real evaluation function can be expressed by this kind of a formula. However, while it seems doubtful that, for example, the optimal policy for chess can be expressed by a function with just a few coefficients, it is entirely possible that a good level of playing can be achieved this way.

The main idea of this approach is not an approximation using fewer coefficients a function, which in fact requires many more of them, but the generalization. This is, we want to generate a policy valid for all the states based on the analysis of a small fraction of them.

For examples, in the experiments with the game backgammon, it was possible to train a player to a level of play comparable to human players based on examining one in $10^{12}$ states.

Obviously, the success in reinforcement learning in such cases depends on the correct selection of the approximation function. If no combination of the selected features can give a good strategy for a game, then no method of learning the coefficients will lead to one. On the other hand, selecting a very elaborate function, with a large number of features and coefficients, increases the chance for a success, but at the expense of a slower convergence and, consequently, the learning process.

## Function parameter correction

In order to facilitate **on-line learning**, some way of updating the parameters based on the reinforcements obtained after each trial (or each step) is needed.

For example, if $u_j(s)$ is the reward-to-go for state $s$ in $j$-th trial, then the utility function approximation error can be computed as:

$$E_j = \frac{(\hat{U}_\theta(s) - u_j(s))^2}{2}$$

The rate of change of this error with respect to the parameter $\theta_i$ can be written as $\partial E_j / \partial \theta_i$, so in order to adjust this parameter toward decreasing the error, the proper adjustment formula is:

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha (u_j(s) - \hat{U}_\theta(s)) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

The above formula is known as the **Widrow-Hoff** or the **delta** rule.

## An example

As an example, for the 4x3 environment the state utility function could be approximated using a linear combination of the coordinates:

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y$$

According to the delta rule the corrections will be given by:

$$\begin{aligned}
\theta_0 &\leftarrow \theta_0 + \alpha(u_j(s) - \hat{U}_\theta(s)) \\
\theta_1 &\leftarrow \theta_1 + \alpha(u_j(s) - \hat{U}_\theta(s))x \\
\theta_2 &\leftarrow \theta_2 + \alpha(u_j(s) - \hat{U}_\theta(s))y
\end{aligned}$$

Assuming for an example $\theta =< \theta_0, \theta_1, \theta_2 >=< 0.5, 0.2, 0.1 >$ we get the initial approximation $\hat{U}_\theta(1, 1) = 0.8$. If, after executing a trial, we computed eg. $u_j(1, 1) = 0.72$, then all the coefficients $\theta_0$, $\theta_1$, $\theta_2$ would be reduced by $0.08\alpha$, which in turn would decrease the error for the state (1,1). Obviously, all the values of $\hat{U}_\theta(s)$ would then change, which is the idea of generalization.

## Application of temporal differences

It is also possible to make updates using the temporal difference method.

$$\theta_i \leftarrow \theta_i + \alpha[R(s) + \gamma \hat{U}_\theta(s') - \hat{U}_\theta(s)] \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

$$\theta_i \leftarrow \theta_i + \alpha[R(s) + \gamma \max_{a'} \hat{Q}_\theta(s', a') - \hat{Q}_\theta(s, a)] \frac{\partial \hat{Q}_\theta(s, a)}{\partial \theta_i}$$

## Useful resources

David Silver: Reinforcement Learning course (2015, University College London):
`http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html`