PIQLE: A Platform for Implementation of Q-Learning Experiments[1]

Francesco De Comité[2]
Samuel Delepoulle[3]

Presentation

PIQLE is a set of Java classes intended to help users quickly implement and test either new reinforcement learning algorithms on benchmark problems, or, looking on the other side, test new problems with standard reinforcement learning algorithms. The clear separations between agents, learning algorithms, world description, and episodes managing, together with Java's inheritance concept makes it easy to focus only on the interesting part of the implementation, for example coding and testing a new problem, testing a new algorithm, or change parameter settings for a known algorithm.
In order to be able to understand what has been learned, $Q(s,a)$ tables can be saved in `arff` format, enabling the researcher to use `Weka` machine learning tools to interpret or to build a model from tabulated Q(s,a) values, using for example decision trees or neural networks.
For problems where the states can be enumerated, Value Iteration Algorithm can also be applied.
For the moment, **PIQLE** does not handle continuous problems, but is able to treat discretized versions of continuous problems (like the Mountain Car problem).
PIQLE can treat one player games or puzzles and two players games. Extensions could be foreseen for multi-agent systems, where each agent could use a different RL algorithm, or different settings of some RL algorithm.
PIQLE, examples of use, and its associated documentation are available at `www.lifl.fr/~decomite/piqle`

General architecture

PIQLE is divided into five main families of classes, each of them communicating with one or more of the other families:

Arbitres : Control the experiments settings (length of episodes), ask the agents to act, collect statistics (length of episodes, total reward...), monitor the output (text or graphics).

Agents : Interface between environment (the problem's implementation) and learning algorithms. To each agent is associated an *algorithm* which will give it advices on the best action to perform. One can view it as a *guru*. An agent, knowing its current state and the list of possible actions, asks its algorithm for the best action to perform. It then informs its environment of this action. In return it receives a new state, and a reward. Those informations are then sent to the algorithm which can use them to learn. An agent can also communicate with its associated `Arbitre`, to inform it of its current state and last performed action. On a more practical point of view, we can save `Agents`, and use them (and what they have learned!) in other programs.

Algorithmes : In this family are gathered all the (learning) algorithms which can, given a state and alist of actions, return what they think to be the best next action. Here one will find classical reinforcement learning algorithms (see below), or can implement experimental ones. An algorithm receives a state and a list of actions, and choose among this list. Classes are defined in such a way that algorithms do not have to know the actual meaning of states and actions, as they only have to store 3-uples $(s,a,Q(s,a))$. Thus are they completely problem-independent.

Environnement : A generic package giving the definition of `states`, `actions`, and `world` (named `Contraintes` in the program). For every problem, the user must describe what is a `state`, an `action`, and how to find all possible actions available from a given state. The `world` also gives rewards, tells when the episode is over, and who won (if this notion has some meaning for the coded problem).

Coded problems : Each problem is an instanciation of the `environnement` classes. Coding `states` and `actions` defines how the agent perceives its world : different coding for the same problem can induce different behaviours, exactly like changing the sensors on a robot (see the maze variations in package `labyrinthe`). Incompletely defining `states` leads to the study of POMDP problems.

[1]Partially supported by project:"ACI Masse De Données"
[2]Grappa,LIFL,UPRESA 8022 CNRS,Université de Lille 1 and Université Lille 3,FRANCE.e-mail:decomite@lifl.fr
[3]LIL,Université du Littoral,Calais,FRANCE.e-mail:delepoulle@lil.univ-littoral.fr

Algorithms

We have implemented, tested, and verified the following algorithms:

Benchmark algorithms : two algorithms serve as lower bounds : a random algorithm, which return a possible action at random in the list, and a *winner*, which plays at random, except if it sees it can win now.

Reinforcement learning : Standard Q-Learning, Watkins's and Peng's $Q(\lambda)$, in their tabulated form and using neural network to memorize $Q(s, a)$ values. All the parameters of those algorithms can be changed using suitable methods, which allows the user to tune them, either by hand, or by program.

Experimental algorithms : the structure of **PIQLE** allows to write new algorithms and then test them on already implemented problems, without having to change anything in the coding of those problems. Two such algorithms are included in the distribution:

- An approximation of Q-Learning using only short integer arithmetic calculations, for use in micro-robots embedded programs.
- A reinforcement learning scheme which use machine learning techniques to learn the $Q(s, a)$ values, either by using our own implementation of feed-forward neural networks, or by calling `Weka` classifiers. Using `Weka` gives access to a wide range of robust implementation of classifiers, for supervised or unsupervised learning, and the structure of **PIQLE** allows to choose the learning algorithm in just one instruction. More, `Weka` graphical tools can be called from inside the programs, giving to the user the possibility of exploring the $Q(s, a)$ data collected by the RL algorithm.

Coded problems

The first goal of **PIQLE** was to redo classical reinforcement learning experiments, so we implemented the Gambler's Problem, Jack's Car Rental, Random Walk, Mountain Car, Acrobot. We also implemented the 'Missionaries and Cannibals' problem, and a general framework for mazes description and resolution. Some two players games are also coded, in particular those for which a winning strategy is known, like Jenga and Memory. We also implemented Othello, but our algorithms do not learn much on that game...
We are now testing a universe where a mobile, with limited visual perception of its environment, must move into a closed area, avoiding obstacles. Results are quite encouraging (see **PIQLE** web page for some screenshots).

Perspectives

Due to the quite complete isolation of each family of classes, we can choose the topic on which we would like to focus, knowing that this would have no side effects on the other parts of the system. Among the current threads, we can cite:

Kohonen Maps : use self-organizing maps to model the environment and store $Q(s, a)$ values. This could be thought like learning a tiling.

Ensemble learning : `agents` will ask a committee of experts for the best action to perform. Good advices must be rewarded...

RL-Framework : in order to participate to the benchmark competition, we think we can define a new `arbitre` and a new `agent`:

- The new `arbitre` will be the interface between agents and RL-Framework.
- The new `agent` will *wrap* its calls to its own algorithm into RL-Framework compatible methods.

We believe that working in that direction, we could easily and quickly propose an interface between both frameworks.