Integration of van Hasselt's RL library into RL-glue

José Luis Benacloch Ayuso
ETSINF, Universitat Politècnica de València, València, Spain.
jobeay@fiv.upv.es

March 15, 2012

Abstract

This paper describes the integration between the library of RL algorithms developed by Hado van Hasselt and RL-glue.

Keywords: Reinforcement learning, RL-glue, SARSA, Q-learning, QV-learning, RL-GGP.

1 Introduction

This note describes how the library of RL algorithms developed by Hado van Hasselt has been integrated into RL-glue. The goal of this integration is more ambitious, since it tries to connect these two things with Jocular[5], a platform which creates players for the GGP (General Game Player) project, using the GGP server [4]. This leads to the name of the project, called RL-GGP. The final motivation of this project is to be able to compare RL algorithms in the context of games, as suggested by some notions of intelligence evaluation using games and social environments [12][10][13][14], which in turn, follow some previous approaches about machine evaluation [1][2][11][6][8][7][9][3].

In this document, we just focus on the integration between RL-glue and Hado van Hasselt's algorithm. In another document, we will deal with the integration between RL-glue and jocular.

2 Integration

We will first describe RL-glue and van Hasselt's RL library. Next we will see their integration and some hints about installation.

2.1 RL-glue

Reinforcement learning (RL) [15] is a relevant area in artificial intelligence which aims at developing methods and algorithms for building agents which make actions in an environment and learn from rewards.

RL-Glue[1] (Reinforcement learning Glue) [16] is a standard interface for the reinforcement learning community developed by Brian Tanner and Adam White. This interface allows researchers and practitioners to connect agents and environments written in very different languages, such as C/C++, Java, Lisp, Matlab or Python, and to design experiments. This is a significant step in the RL community, since a lot of work is usually devoted whenever new algorithms are implemented and they need to be compared or connected to other systems. For instance, RL-Glue is being used

[1]The webpage of the project is `http://glue.rl-community.org/wiki/Main_Page`.

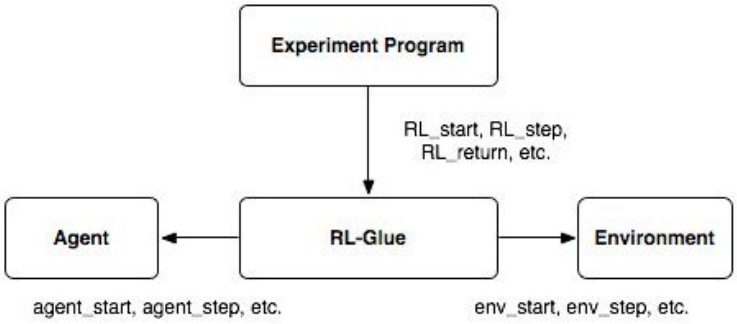

Figure 1: Schematic view of how the Experiment Program, the Agent and the Environment modules connect in RL-Glue. Image taken from the RL-Glue webpage.

in the now regular "Reinforcement learning competition" [18]. Figure 1 shows a schematic view of how RL-Glue connects agents and environments, and experiments.

One of the current limitations of RL-glue is that its RL-Library is still quite limited. While the number of environments to play with is increasing, the number of algorithms is very small. At present, it only includes a random agent (RandomAgentJava) and a version of SARSA (Tile Coding Sarsa Lambda Java).

2.2 van Hasselt's RL library

In the context of the experiments related to his PhD [17], Hado van Hasselt has developed a complete library of RL algorithms[2], available in C/C++ and Python. The library contains the algorithms ACLA (Actor-Critic Learning Automaton), CACLA (Continuous Actor-Critic Automaton), Q-learning, QV-learning and SARSA.

The emphasis is put on working on either continuous or discrete states (e.g. CACLA is a continuous version of ACLA), so these are really general implementations of some of these algorithms. In our integration, however, we restrict our focus to discrete state algorithms: Q-learning, QV-learning and SARSA. In what follows, we assume states are discrete.

Because of a previous integration with other systems outside RL-glue that were in Java (the system Jocular), we initially migrated Hasselt's RL library to Java, and then we kept with that language for the rest of the project. We then use the RL-Glue interface in Java.

There is a class known as `algorithm` which is used to define a RL algorithm. This class contains three variables:

- The Q matrix where we store the Q-values according to the state and action. This matrix is bidimensional (`double Q[][]`), where the first element is the state index and the second one is action.
- The number of states and number of actions, defined as `int numberOfStates`, and `numberOfActions`, respectively.

The class `Algorithm` requires two classes to work, namely `State` and `Action_`, which are used for representing states and actions, respectively. The class `State` consists of a field with the number of total states `int numberOfStates` and a field with the current state, `int discreteState`. Similarly,

[2]Available at `http://homepages.cwi.nl/~hasselt/code.html`.

the class `Action_` consists of a field with the number of total actions `int numberOfActions` and a field with the current action, `int discreteAction`.

The methods in the class `algorithm` are basically those which choose an action given a state and the Q matrix

- The first action getting the maximum in Q matrix for a given state is given by the method `void getMaxActionFirst( State state, Action_ action )`.
- A random action in the Q matrix for a given state is produced by the method `void getMaxActionRandom( State state, Action_ action )`.
- We can also use the *boltzmann* or *egreedy* methods to calculate the best action, using method `void boltzmann( State state, Action_ action, double tau )` and `void egreedy( State state, Action_ action, double epsilon )`.

There are more variables and methods but we will not use them in the current connection with RL-glue.

The implementation of the algorithms is done through the extension of the class `Algorithm`. Several methods must be written for making it work, namely:

- The constructor needs the number of states and actions, and whether they are discrete or continuous, in order to initialise the Q matrix. These data are embedded in the class `World_` through the fields `numberOfActions, numberOfStates`. The class `World_` is created by default when states are discrte. For instance, the constructor for the SARSA algorithm is `Sarsa ( World_ w )`.
- A method to increase the size of the Q matrix in case the number of states or actions increases (`update_size_Q(int n_s, int n_a)`).
- Two methods to store and retrieve the Q matrix from file. This has been done to be able to use the acquired 'knowledge' in the Q matrix from a game session (episode) to a other sessions. These are `void saveValueFunction(String theFileName)` and `boolean loadValueFunction(String theFileName)`. This method is loaded whenever a constructor initialises the Q matrix.

In addition, there is a series of methods which have to be modified depending on the algorithm:

- There is one method (`String getName()`) which just resturns the name of the algorithm. This method is important since it constructs the file where the matrix is loaded and stored using this name, by adding the extension ".csv".
- The method which updates the Q matrix (and which learns) is defined as `void update( State state, Action_ actions[], double rt, State nextState, boolean endOfEpisode, double learningRate[], double gamma )`. The parameters are the previous state, an array with the previous action and the current action, the current states, whether it has reached the goal or the end of an episode, an array with learning factors and, finally, the gamma factor. It is important to note that the last instruction included in the previous method *must* be: "`saveValueFunction(getName()+".csv");`", which stores the Q matrix at the end of the interaction (or episode).
- Finally, there is a method which returns the size of the array (`double learningRate[]`), which is defined as `int getNumberOfLearningRates()`.

2.3 Integration into RL-glue

The connection requires that the RL-Glue interface in Java embeds the `Algorithm` class which is defined in van Hasselt's RL library. In order to do this, the algorithms were first migrated to Java to be included in RL-Glue using the Jave RL-Glue interface.

After the migration, we need that RL-Glue calls the methods in the class `Algorithm` above, which choose the action depending on the used algorithm.

In this way, the agent in RL-Glue really makes the decisions by just calling to the code integrated from Hasselt's RL library.

Basically, we have added an agent initialisation part (method `agent_start`). The first action will be taken according to previous actions if there is a file from previous games. There is also a file: "config_agent.cfg" which just contains an integer which is used to select the algorithm to be used (0: Sarsa, 1: Q-learning, 2: QV-learning).

```
// Initialises the attribute w of type World_ for all the algorithms
   w = new World_();

   w.numberOfActions = num_action;
   w.numberOfStates = observation.intArray[1];

// Reads the algorithm to be used from file and stores this in funcion_apr
   loadConfigAgent("config_agent.cfg");

// Initalises state
   State s = new State();
   s.discrete =true;
   s.discreteState = observation.intArray[0];

// Initialises action array
   actions = new Action_[2];
   actions[0] = new Action_();
   actions[0].discrete = true;
   actions[1] = new Action_();
   actions[1].discrete = true;

// Gets the action depending on the algoritm
    switch(funcion_apr){
      case 0:
       sarsa = new Sarsa_(w);
       sarsa.getMaxActionFirst(s, actions[0]);
       break;
      case 1:
       QL = new QLearning_(w);
       QL.getMaxActionFirst(s, actions[0]);
       break;
      case 2:
       QVL = new QVLearning_(w);
       QVL.getMaxActionFirst(s, actions[0]);
```

```
      break;
    default:
      System.err.println("ERROR. The agent algorithm is not known");
      break;
  }
```

The options are SARSA, QL and QVL, which are the RL algorithms implemented in van Hasselt's library. We can see that the method `getMaxActionFirst` is used to get the action with greatest value for each algorithm, depending on the current state.

Each an agent is asked to make an action (method `agent_step`), this method will call the methods in the `algorithm` class in order to get the action which gets highest value in the matrix, as well as the method that updates the matrix size (if necessary) and, finally, the method which updates the matrix content.

```
// Gets the current action according to algorithm, Q matrix and state
// It also updates the size of Q (if needed) and its content

switch(funcion_apr) {
  case 0:
    sarsa.update_size_Q(observation.getInt(1), num_action);
    sarsa.getMaxActionFirst(newS, actions[0]);

    theIntAction = actions[0].discreteAction;

    actions[0].discreteAction = lastActionInt;
    actions[1].discreteAction = theIntAction;

    learning_rat = new double[sarsa.getNumberOfLearningRates()];

    for(int i=0; i<sarsa.getNumberOfLearningRates(); i++)
       learning_rat[i] = learning_r;

    sarsa.update(lastS, actions, reward, newS, false, learning_rat, 1);
    break;

  case 1:
     QL.update_size_Q(observation.getInt(1), num_action);
     QL.getMaxActionFirst(newS, actions[0]);

// ........

  default:
    System.err.println("ERROR. The agent algorithm is not known");
    break;
}
```

Finally, when the game ends, the agent has a special method (`agent_end`) given the last action and state. There must a `true` argument to indicate that it is the last time that the class `algorithm` is updated, as follows (illustrated for SARSA):

```
sarsa.update(lastS, actions, reward, lastS, true, learning_rat, 1);
```

For episodes where we want to propagate a bad reward when the agent loses (games) or any other negative outcome after a complete episode, there is a method `lose` which updates the algorithm. The negative reward is an argument of this method.

3 Installation and final remarks

The RL-GGP system integrates the RL algorithms developed by Hado van Hasselt into RL-glue, but also connect these two things with Jocular[5], a platform which creates players for the GGP (General Game Player) project, using the GGP server [4].

The installation of the whole system can be done from:

`http://users.dsic.upv.es/~flip/RLGGP/`

where the part which just integrates RL algorithms into RL-glue can be easily separated.

There are of course, several things that can be improved in the implementation. For instance, the file "config_agent.cfg" could also include the parameters of each algorithm, which currently can only be modified by program.

Acknowledgements

We thank Hado van Hasselt for using his implementation and the RL-community for developing RL-glue. This document has been extracted, translated and summarised from Benacloch's project by José Hernández-Orallo.

References

[1] D. L. Dowe and A. R. Hajek. A computational extension to the Turing Test. *in Proceedings of the 4th Conference of the Australasian Cognitive Science Society, University of Newcastle, NSW, Australia*, 1997.

[2] D. L. Dowe and A. R. Hajek. A non-behavioural, computational extension to the Turing Test. In *Intl. Conf. on Computational Intelligence & multimedia applications (ICCIMA'98), Gippsland, Australia*, pages 101–106, 1998.

[3] D. L. Dowe and J. Hernandez-Orallo. IQ tests are not for machines, yet. *Intelligence*, 40(2):77–81, 2012.

[4] M. Genesereth, N. Love, and B. Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62, 2005.

[5] D. Haley. Jocular. *http://games.stanford.edu/resources/reference/jocular/jocular.html*.

[6] J. Hernández-Orallo. Beyond the Turing Test. *J. Logic, Language & Information*, 9(4):447–466, 2000.

[7] J. Hernández-Orallo. Constructive reinforcement learning. *International Journal of Intelligent Systems*, 15(3):241–264, 2000.

[8] J. Hernández-Orallo. On the computational measurement of intelligence factors. In A. Meystel, editor, *Performance metrics for intelligent systems workshop*, pages 1–8. National Institute of Standards and Technology, Gaithersburg, MD, U.S.A., 2000.

[9] J. Hernández-Orallo. Thesis: Computational measures of information gain and reinforcement in inference processes. *AI Communications*, 13(1):49–50, 2000.

[10] J. Hernández-Orallo and D. L. Dowe. Measuring universal intelligence: Towards an anytime intelligence test. *Artificial Intelligence*, 174(18):1508 – 1539, 2010.

[11] J. Hernández-Orallo and N. Minaya-Collado. A formal definition of intelligence based on an intensional variant of Kolmogorov complexity. In *Proc. Intl Symposium of Engineering of Intelligent Systems (EIS'98)*, pages 146–163. ICSC Press, 1998.

[12] B. Hibbard. Adversarial sequence prediction. In *Proceeding of the 2008 conference on Artificial General Intelligence 2008: Proceedings of the First AGI Conference*, pages 399–403. IOS Press, 2008.

[13] J. Insa-Cabrera, D. L. Dowe, S. España-Cubillo, M. V. Hernández-Lloreda, and J. Hernández-Orallo. Comparing humans and AI agents. In J. Schmidhuber, K.R. Thórisson, and M. Looks, editors, *Artificial General Intelligence*, volume 6830, pages 122–132. LNAI, Springer, 2011.

[14] J. Insa-Cabrera, D. L. Dowe, and J. Hernandez-Orallo. Evaluating a reinforcement learning algorithm with a general intelligence test. In J.A. Moreno J.A. Lozano, J.A. Gamez, editor, *Current Topics in Artificial Intelligence. CAEPIA 2011*. LNAI Series 7023, Springer, 2011.

[15] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

[16] B. Tanner and Ad. White. RL-Glue : Language-independent software for reinforcement-learning experiments. *Journal of Machine Learning Research*, 10:2133–2136, September 2009.

[17] H.P. van Hasselt. Insights in reinforcement learning: formal analysis and empirical evaluation of temporal-difference learning algorithms. *SIKS dissertation series*, 2011(04), 2011.

[18] S. Whiteson, B. Tanner, and A. White. The Reinforcement Learning Competitions. *The AI magazine*, 31(2):81–94, 2010.