Java Resources for Teaching Reinforcement Learning

Amy J. Kerr, Todd W. Neller, Christopher J. La Pilla, and Michael D. Schompert
Gettysburg College
Department of Computer Science
Campus Box 402
Gettysburg, PA 17325
E-mail: tneller@gettysburg.edu

Abstract— **In this paper we present a library of classes for programming reinforcement learning simulations in Java. This library is based upon the standard by Sutton and Santamaria [1], with valuable additions to simplify the implementation of the software for selected temporal-difference control algorithms and various memory models. We also present arguments for the integration of this library into the curriculum of a Java-based undergraduate course in Artificial Intelligence.**

I. Introduction

Reinforcement learning (RL) is a form of machine learning in which a computational agent learns entirely through experience, by trying actions and analyzing the consequences of these actions. The agent's actions typically bear both immediate and delayed consequences. In the immediate sense, the agent receives a numerical reward for each action. As shown in Figure1, each action also affects the situations and available choices that follow for the agent. That is, each action causes a transition into a subsequent state of the agent's environment and thus affects the agent's opportunities for future rewards. In this way, actions also have delayed, indirect rewards. This combination of trial-and-error and delayed rewards yields the intriguing complexity of RL problems. For a complete introduction to RL and the algorithms mentioned hereafter, we recommend [2], [3], [4]. For the purpose of this paper, we will assume an understanding of basic RL theory and algorithms.

The RL problem can be formalized by the interaction of two basic entities: the *agent* and the *environment*. The agent is the learner and decision-maker. The agent's environment is comprised of everything that it cannot completely control. Thus, the environment defines the task that the agent is seeking to learn. A third entity, the *simulation*, mediates the interactions between the agent and the environment.

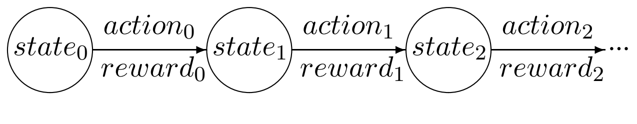

Fig. 1. Each action generates an immediate numerical reward and a state transition.

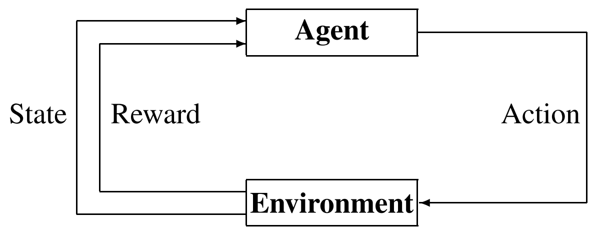

Fig. 2. The agent-environment interaction in reinforcement learning.

The interactions between the agent and the environment fall into three categories: *actions*, *rewards*, and *state descriptions*. As depicted in Figure 2, the agent takes actions. In response to each action, the environment outputs a reward, the immediate numerical payoff for the action. Additionally, the environment transitions into a subsequent state and relays some information about this new state to the agent.

The three basic entities of the RL problem and the three forms of communication between the agent and the environment provide a concise outline for the RL problem and for the fundamental classes and interfaces of the standard interface for programming RL software.

II. The Java Reinforcement Learning Library

In this paper, we present a library of classes for implementing reinforcement learning in Java. The core component of this library is the standard Java interface for programming RL problems. This Java interface is an adaptation of the C++ standard [5] of Sutton and Santamaria's RL interface [1]. Like the C++ version, the Java standard is comprised of three core classes, which correspond to the three basic entities of the RL problem. These three classes are `Agent`, `Environment`, and `Simulation`. These classes, and their key methods, are outlined in Figure 3.

The presented library includes valuable supplemental code as well. This code includes interfaces and classes for implementing the communication between agents and environments. The library contains `Action` and `State` interfaces for this purpose. A wrapper class, `ActionResult`, is also provided for concisely describing the environment's response to an agent's action; an `ActionResult` instance contains a `double` reward value and a `State` instance.

Perhaps the most useful additions to the standard interface are the provided `Agent` subclasses for programming agents that employ temporal-difference (TD) learning. These subclasses are easily adaptable to most (if not all) possible memory models for agents. Selected TD control algorithms are implemented in full for agents with tabular forms of memory. Code is also supplied for a multilayer feedforward neural network with backpropagation, which is one common memory model.

The library also includes a complete implementation of a RL problem to demonstrate the use of the contained classes. The example implementation is for a simple maze game adapted from [2]'s Grid World example. The package contains `Agent` subclasses that employ various TD learning-based algorithms (including SARSA and Watkin's Q-Learning) with various memory models (including neural networks and look-up tables).

The complete Javadoc documentation for the components of the presented library can be found at `http://cs.gettysburg.edu/projects/javaRL`.

The presented standard interface can serve as the kernel of any RL software. Thus, we now refocus our attention on the three core classes of the interface. We briefly outline the three primary methods that appear in each of these classes, pausing briefly along the way to highlight the jewel of the library, the `Agent` subclasses for implementing TD learning.

A. The `Agent` *Class*

The `Agent` class is one of the three core classes of the standard Java interface. It is the abstract class for implementing all RL agents. A RL agent interacts with an environment by selecting actions and consequently receiving numerical rewards and state descriptions from the environment (Figure 2). The immediate numerical reward quantifies the short-term desirability of an action; the `State` describes the new state of the environment and thus the potential for future rewards. Ideally, the agent learns from these consequences to select the actions that are most desirable in the *long*-term. By learning this optimal mapping, or *policy*, from each state of the environment to the action that yields the greatest reward in the long-term, the agent learns to optimally complete the assigned task.

The `Agent` class contains three abstract methods. To tailor the standard interface to a specific RL problem, a user must implement an `Agent` subclass and define these three abstract methods: `init`, `startTrial`, and `step`.

The `init` method initializes an agent. This initialization typically requires the construction of one or more data structures for the agent's "memory." In our adaptation of C++ interface by [5], this method has been endowed with the capability of receiving a preconstructed memory structure as a parameter. This feature allows multiple agents to share the same memory in a form of "self-play." The `init` method can either preload past memory or reset the agent's memory to its original, naïve condition. This method is called once when the RL simulation first begins.

The `startTrial` method prepares the agent for the start of a new trial (i.e. episode) within an ongoing simulation. The method must create and return the first action of the agent in the trial. This first action is determined by the agent's current policy and the first state of the trial, which is passed as a parameter to the method. The `startTrial` method is called at the beginning of each trial.

The `step` method is the third abstract method of the `Agent` class. It is the method where all learning and decision-making occurs for an agent. As parameters, the method receives a `State` instance and a `double` reward, which describe the consequences of taking the previous action. If the agent learns in any way with experience, then this method implements this learning. The method then returns the agent's next action, as determined by the agent's current policy. This method is called once on each step of the simulation.

This abstract structure of the `Agent` class provides significant flexibility in the design of RL agents. `Agent` subclasses may implement agents that employ any type of memory model, such as look-up tables, neural networks, or sparse-distributed memories. Furthermore, such subclasses can implement any RL algorithm (in the

```
public abstract class Agent
    public abstract void init(Object[] arr)
    public abstract Action startTrial(State state)
    public abstract Action step(State nextState, double reward)

public abstract class Environment
    public abstract void init(Object[] arr)
    public abstract State startTrial()
    public abstract ActionResult step(Action act)

public abstract class Simulation
    public Simulation(Agent[] agt, Environment env)
    public void init(Object[] arr)
    public void startTrial()
    public void step(boolean collectData)
    public void steps(long numSteps)
    public void trials(long numTrials, long maxStepsPerTrial, int printDivisor)
    public abstract void collectData(State state, Action act,
                                     State nextState, double reward)
```

Fig. 3. An outline of the three core classes of the Java standard.

`step` method), such as Monte Carlo or TD learning.

1) The `TDAgent` *Class:* Some of the most widely-used RL algorithms are based on temporal-difference (TD) learning. One particular algorithm, known as TD(λ), is especially favored by researchers because it seamlessly unifies two prominent RL algorithms, TD(0) and Monte-Carlo. It also offers an efficient solution to the credit assignment problem through its use of eligibility traces. Because of the prominence of this algorithm, the presented library includes an `Agent` subclass, `TDAgent`, for implementing agents that employ TD(λ)-based algorithms. In particular, this subclass is conducive to implementing algorithms that approximate *action-value,* or *Q, functions* using TD methods. Such algorithms include SARSA(λ) and Watkin's Q(λ)-Learning.

The `TDAgent` class includes the common features of most (if not all) TD(λ)-based algorithms. The class contains common constants, as well as seven simple abstract methods to handle action-value updates. These methods are all abstract to maintain flexibility in the types of memory models employable by the agent. Minimal definitions of the `init`, `startTrial`, and `step` methods from the `Agent` class are implemented in `TDAgent` using these seven abstract methods. `TDAgent` also provides code for implementing ϵ-greedy policies, which are the most common RL policies (due to their simple balancing of exploration and exploitation).

The structure of `TDAgent`, with its heavy dependence on abstract accessor and mutator methods, makes the code easily adaptable to most (if not all) TD(λ)-based algorithms and memory models. Furthermore, due to strong resemblances among most action-value adaptations of the TD(λ) algorithm, once a memory model is selected for a given RL problem, only one of `TDAgent`'s methods, `getTDError`, is algorithm-dependent. The presented library factors out the common code for TD(λ)-based agents with tabular memories (i.e. look-up tables) in the `TDAgent` subclass `TDAgentTab`. The library also includes subclasses of `TDAgentTab` with appropriate implementations of `getTDError` for the SARSA(λ) and Watkin's Q(λ)-Learning algorithms.

B. *The* `Environment` *Class*

In RL, the environment is the entity that the agent interacts with and seeks to learn about. The environment responds to an agent's action by outputting a numerical reward and a description of the next environmental state (Figure 2). The `Environment` class provides a means of implementing all instances of RL environments.

The `Environment` class contains the same three abstract methods as the `Agent` class. To tailor the standard interface to a specific RL problem, a user must implement an `Environment` subclass and define its three abstract methods: `init`, `startTrial`, and `step`.

The `init` method initializes an instance of an `Environment` subclass. The method is responsible for constructing and initializing any necessary data structures. If necessary, it can load initial conditions, or, in cases where the environment is adaptive, can reset the environment to its original, naïve condition. This method is called when the simulation first begins.

The `startTrial` method must prepare the environment for the start of a new trial (i.e. episode) within an ongoing simulation. The method must create and return the first `State` of the new trial. This method is called at the beginning of each trial.

The `step` method is the third abstract method in the `Environment` class. This method receives the agent's most recent action as a parameter and it returns the immediate reward for this action, as well as a description of the new state of the environment. This method is called once on each step of the simulation.

C. *The* `Simulation` *Class*

The `Simulation` class is the third and final abstract class of the standard Java interface. Instances of this class manage the interactions between an agent and an environment. This class contains complete implementations of the `init`, `startTrial`, and `step` methods for simulations. Each of these methods calls the `Agent` and `Environment` instances' versions of the same method, often using the return value of one of the calls as the parameter for the other. The classes also contain methods that run the simulation for a set number of steps or trials. To employ this class, a user only needs to define the one abstract method, `collectData`, which periodically collects and potentially prints out data from the simulation for logging purposes.

III. The Value of Using the Java Library in an AI Course

The presented library is a natural addition to a RL module in any Artificial Intelligence (AI) course in which the students are familiar with Java. The RL standard, when incorporated into course assignments, offers the following benefits:

Clarifies the assignment by explicating the task in terms of implementable modules. The class and method names of the standard's structure offer a common vocabulary for use in the assignment description and discussions.

Aids students in structuring their approaches to the assignment by dissecting the programming into conquerable modules.

Facilitates teamwork. The modularity of the code, combined with the (supplied) explicit specifications, is conducive to group projects. This feature increases the feasibility of more complex and intriguing assignments while also providing opportunities for students to refine their cooperative programming skills.

Clarifies RL theory. The standard formalizes the three basic entities of RL by separating them into distinct classes. Furthermore, the interaction between these three entities is explicitly highlighted through the use of the *same* three methods, thus summarizing the cause-and-effect nature of the interactions between the environment and the agent in each step of a simulation.

Highlights TD(λ) learning. Through the variation of the λ parameter, the provided TD(λ)-based code offers experience studying TD(0) and Monte Carlo learning. The supplied code also highlights the commonalities between TD agents, as well as the defining differences, which tend to be slight (i.e. 6 lines of code), between TD(λ)-based algorithms.

Offers flexibility in assignment design. By simply varying the elements of the library (specifications and/or code) that are supplied to the students, the library yields an array of potential assignments of varying foci and complexities. These assignments range from straightforward implementations of the three core classes (for a specific RL task, algorithm, and memory model), to experiments with parameter values (i.e. λ in the TD-subclasses), to comparative studies of various algorithms and memory models. The library makes even complex projects feasible by expediting the coding of the foundational and TD-specific classes (through provided specifications and/or code).

Provides a fully implemented RL problem for use as an example of RL or as an answer key to an identical assignment.

Simplifies grading and correction. By imposing structure on the students' coding, the RL standard generates more uniform code, and it provides a natural, objective grading rubric. The structure also facilitates modular testing for assessment.

In Chapter 2 of [2], a simple n-armed bandit problem is described. We began a special topic course on Machine Learning with a number of exercises related to this problem. Given a complete implementation with an ϵ-greedy agent, students were asked to focus on possible modifications. Students were given choices of various n-armed bandit projects: comparison of the (1) softmax, (2) reinforcement comparison, and (3) pursuit policies with the ϵ-greedy policy regarding average rewards and optimality.

In the next assignments, students were given the task of implementing the extended Environment classes for various problems (e.g. the Gambler's Problem of [2], a simplification of the dice game Pig) for a given Agent specification. Once comfortable with the entire framework, the students were well prepared to delve into the more complex array of Monte Carlo and TD algorithms.

IV. Conclusion

The presented Java library for programming RL software is a natural addition to a RL module in any Artificial Intelligence course. The library, with its standard interface, TD-specific subclasses, fully implemented example of a RL problem, and complete documentation, is a valuable educational tool. Using the interface as a foundation, the library inspires a diverse array of programming assignments, ranging from straightforward implementations of a RL problem to comparative studies and cumulative projects. Through the simple, base requirement of implementing the supplied interface, these assignments provide objective grading rubrics, approachable design, educational summaries of theory, and experience with cooperative software development. The incorporation of the library into an Artificial Intelligence course is beneficial for both the instructor and the students.

References

[1] R. S. Sutton and J. C. Santamaria, "A standard interface for reinforcement learning software," Available: `http://www-anw.cs.umass.edu/~rich-/RLinterface/RLinterface.html`.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: an introduction*. Cambridge, Massachussetts: MIT Press, 1998.

[3] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Englewood Cliff, New Jersey: Prentice-Hall, Inc., 1995, ch. 20.

[4] T. M. Mitchell, *Machine Learning*. Boston, Massachusetts: McGraw-Hill Companies, Inc., 1997, ch. 13.

[5] R. S. Sutton and J. C. Santamaria, "A standard interface for reinforcement learning software in C++, version 1.1," Available: `http://www-anw.cs.umass.edu/~rich-/RLinterface/RLI-Cplusplus.html`.