Journal of Machine Learning Research 14 (2013) 625-628 Submitted 9/11; Revised 6/12; Published 2/13

A C++ Template-Based Reinforcement Learning Library: Fitting the Code to the Mathematics

Hervé Frezza-Buet[*] HERVE.FREZZA-BUET@SUPELEC.FR
Matthieu Geist MATTHIEU.GEIST@SUPELEC.FR
Supélec
2 rue Édouard Belin
57070 Metz, France

Editor: Mikio Braun

Abstract

This paper introduces the `rllib` as an original `C++` template-based library oriented toward value function estimation. Generic programming is promoted here as a way of having a good fit between the mathematics of reinforcement learning and their implementation in a library. The main concepts of `rllib` are presented, as well as a short example.

Keywords: reinforcement learning, `C++`, generic programming

1. `C++` Genericity for Fitting the Mathematics of Reinforcement Learning

Reinforcement learning (RL) is a field of machine learning that benefits from a rigorous mathematical formalism, as shown for example by Bertsekas (1995). Although this formalism is well accepted in the field, its translation into efficient computer science tools has surprisingly not led to any standard yet, as mentioned by Kovacs and Egginton (2011). The claim of this paper is that genericity enables a natural expression of the mathematics of RL. The `rllib` (2011) library implements this idea in the `C++` language, where genericity relies on templates. Templates automate the re-writing of some generic code involving user types, offering a strong type checking at compile time that improves the code safety.

Using the `rllib` templates requires that the user-defined types fit some documented concepts. For example, some class `C` defining an agent should be designed so that `C::state_type` is the type used for the states, `C::action_type` is the type used for the actions, and the method `C::action_type policy(const C::state_type& s) const` is implemented, in order to compute the action to be performed in a given state. This concept definition specifies what is required for an agent mathematically. Note that `C` does not need to inherit from any kind of abstract `rl::Agent` class to be used by the `rllib` tools. It can be directly provided as a type argument to any `rllib` template requiring an argument fitting the concept of an agent, so that the re-written code actually compiles.

2. A Short Example

Let us consider the following toy-example. The state space contains values from 0 to 9, and actions consist in increasing or decreasing the value. When the value gets out of bounds, a reward is returned

[*]. Also at UMI 2958 Georgia Tech / CNRS, 2-3, rue Marconi, 57070 Metz, France.

(-1 for bound 0, 1 for bound 9). Otherwise, a null reward is returned. Let us define this problem and run Sarsa. First, a simulator class fitting the concept `Simulator` described in the documentation is needed.

```
class Sim { // Our simulator class. No inheritance required.
private:
  int current; double r;
public:
  typedef int            phase_type  ; typedef int    observation_type;
  typedef enum {up,down} action_type ; typedef double reward_type;

  Sim(void) : current(0), r(0) {}
  void setPhase(const phase_type &s)  {current = s%10;}
  const observation_type& sense(void) const {return current;}
  reward_type reward(void) const {return r;}
  void timeStep(const action_type &a) {
    if(a == up) current++; else current--;
    if(current < 0) r=-1; else if(current > 9) r=1; else r=0;
    if(r != 0) throw rl::exception::Terminal("Out_of_range");
  }
};
```

Following the concept requirements, the class `Sim` naturally implements a sensor method `sense` that provides an observation from the current phase of the controlled dynamical system, and a method `timeStep` that computes a transition consecutive to some action. Note the use of exceptions for terminal states. For the sake of simplicity in further code, the following is added.

```
typedef Sim::phase_type                       S;
typedef Sim::action_type                      A;
typedef rl::Iterator<A,Sim::up,Sim::down> Aenum; // enumerate all actions.
typedef rl::SA<S,A>                           SA;
typedef rl::sa::Transition<S,A,double>     Transition; // i.e., (s,a,r,s',a')
```

As Sarsa computes some Q-values, a structure is needed to store these values. A 10×2 array could be used, but our library is oriented toward value function estimation. It means that the Q-function computed by Sarsa is an element taken in some parametrized set of functions from $S \times A$ to $\mathbb{R}$. The parameter used is a vector. Such Q-function fits the concept of `DerivableArchitecture`. Let us here consider a tabular representation of Q: a $|S| \times |A| = 20$ array is actually implemented, but it is viewed as some particular case of parametrized function (there is one dimension in the parameter vector for each Q-value in the array, that is, the parameter vector *is* the array). This leads to the following definition of `Q`. The explicit definition of the gradient of `Q` according to the parameter is required.

```
class Q { // Tabular representation of Q (theta contains the Q[s,a]).
public:
  typedef S           state_type          ; typedef A          action_type;
  typedef SA          sa_type             ; typedef SA         input_type;
  typedef Transition  sa_transition_type  ; typedef Transition transition_type;
  typedef gsl_vector* param_type          ; typedef double     output_type;

  param_type newParameterInstance(void) const {
    return gsl_vector_calloc(20); // 20 = |S|*|A|
  }
  output_type operator()(const param_type theta,
                         state_type s, action_type a) const {
    return gsl_vector_get(theta,2*s+a); // return the value q_theta(s,a).
  }
  void gradient(const param_type theta,
                state_type s,action_type a,
                param_type grad) const {
```

```
    gsl_vector_set_basis(grad,2*s+a); // ..001000... with 1 at [s,a]
  }
};
```

The simulator and the parametrized Q-function are the only classes to be defined, since they are problem-dependent. From these types, the Sarsa algorithm can be easily implemented from `rllib` templates, as well as different kinds of agents. Here, learning is performed by an ε-greedy agent, while testing is executed by a greedy agent.

```
class Param {
public:
  static double gamma(void)   {return .99;}
  static double alpha(void)   {return .05;}
  static double epsilon(void) {return 0.2;}
};
typedef rl::sa::SARSA<Q,Param>                                    Critic;
typedef rl::sa::ArgmaxFromAIteration<Critic,Aenum>                ArgmaxCritic;
typedef rl::agent::Greedy<ArgmaxCritic>                           TestAgent;
typedef rl::agent::online::EpsilonGreedy<ArgmaxCritic,Aenum,Param> LearnAgent;
```

The `rllib` expresses that Sarsa provides a critic, offering a Q-function. As actions are discrete, the best action (i.e., $\text{argmax}_{a\in A} Q(s,a)$) can be found by considering all the actions sequentially. This is what `ArgmaxCritic` offers thanks to the action enumerator `Aenum`, in order to define greedy and ε-greedy agents. The main function then only consists in running episodes with the appropriate agents.

```
int main(int argc, char* argv[]) {
  Sim          simulator;          // This is what the agent controls.
  Transition   transition;         // This is some s,a,r,s',a' data.
  ArgmaxCritic critic;             // This computes Q and argmax_a Q(s,a).
  LearnAgent   learner(critic);    // SARSA uses this agent to learn the policy.
  TestAgent    tester(critic);     // This behaves according to the critic.
  A            a;                  // Some action.
  S            s;                  // Some state.
  int          episode,length,step=0;

  for(episode = 0; episode < 10000; ++episode) { // Learning phase
    simulator.setPhase(rand()%10);
    rl::episode::sa::run_and_learn(simulator,learner,transition,0,length);
  }
  try { // Test phase
    simulator.setPhase(0);
    while(true) {
      s = simulator.sense(); a = tester.policy(s);
      step++; simulator.timeStep(a);
    }
  }
  catch(rl::exception::Terminal e) {std::cout << step << " steps." << std::endl;}
  return 0; // the message printed is ``10 steps.''
}
```

3. Features of the Library

Using the library requires to define the features that are specific to the problem (the simulator and the Q-function architecture in our example) from scratch, but with the help of concepts. Then, the specific features can be handled by generic code provided by the library to implement RL techniques with value function estimation.

Currently, Q-learing, Sarsa, KTD-Q, LSTD, and policy iteration are available, as well as a multi-layer perceptron architecture. Moreover, some benchmark problems (i.e., simulators) are also provided: the mountain car, the cliff walking, the inverted pendulum and the Boyan chain. Extending the library with new algorithms is allowed, since it consists in defining new templates. This is a bit more technical than only using the existing algorithms, but the structure of existing concepts helps, since it reflects the mathematics of RL. For example, concepts like `Feature`, for linear approaches mainly (i.e., $Q(s,a) = \theta^{\mathrm{T}}\varphi(s,a)$) and `Architecture` (i.e., $Q(s,a) = f_\theta(s,a)$ for more general approximation) orient the design toward functional approaches of RL. The algorithms implemented so far rely on the GNU Scientific Library (see `GSL`, 2011) for linear algebra computation, so the `GPL` licence of `GSL` propagates to the `rllib`.

4. Conclusion

The `rllib` relies only on the `C++` standard and the availability of the `GSL` on the system. It offers state-action function approximation tools for applying RL to real problems, as well as a design that fits the mathematics. The latter allows for extensions, but is also compliant with pedagogical purpose. The design of the `rllib` aims at allowing the user to build (using `C++` programming) its own experiment, using several algorithms, several agents, on-line or batch learning, and so on. Actually, the difficult part of RL is the algorithms themselves, not the script-like part of the experiment where things are put together (see the main function in our example). With a framework, in the sense of Kovacs and Egginton (2011), the experiment is not directly accessible to the user programs, since it is handled by some libraries in order to offer graphical interface or analyzing tools. The user code is then called by the framework when required. We advocate that allowing the user to call the `rllib` functionality at his/her convenience provides an open and extensible access to RL for students, researchers and engineers.

Last, the `rllib` fits the requirements expressed by Kovacs and Egginton (2011, Section 4.3): support of good scientific research, formulation compliant with the domain, allowing for any kind of agents and any kind of approximators, interoperability of components (the Q function of the example can be used for different algorithms and agents), maximization of run-time speed (use of `C++` and templates that inline massively the code), open source, etc. Extensions of `rllib` can be considered, for example for handling POMDPs, and contributions of users are expected. The use of templates is unfortunately unfamiliar to many programmers, but the effort is worth it, since it brings the code at the level of the mathematical formalism, increasing readability (by a rational use of typedefs) and reducing bugs. Even if the approach is dramatically different from existing frameworks, wrappings with frameworks can be considered in further development.

References

Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. Athena Scientific, 3rd (2005-2007) edition, 1995.

`GSL`, 2011. `http://http://www.gnu.org/software/gsl`.

Tim Kovacs and Robert Egginton. On the analysis and design of software for reinforcement learning, with a survey of existing systems. *Machine Learning*, 84:7–49, 2011.

`rllib`, 2011. `http://ims.metz.supelec.fr/spip.php?article122`.