RLLib: C++ Library to Predict, Control, and Represent Learnable Knowledge Using On/Off Policy Reinforcement Learning

Saminda Abeyruwan(✉) and Ubbo Visser

Department of Computer Science, University of Miami, 1365 Memorial Drive, Coral Gables, FL 33146, USA
{saminda,visser}@cs.miami.edu

Abstract. RLLib is a lightweight C++ template library that implements incremental, standard, and gradient temporal-difference learning algorithms in reinforcement learning. It is an optimized library for robotic applications and embedded devices that operates under fast duty cycles (e.g., $\leq$30 ms). RLLib has been tested and evaluated on RoboCup 3D soccer simulation agents, NAO V4 humanoid robots, and Tiva C series launchpad microcontrollers to predict, control, learn behavior, and represent learnable knowledge.

Keywords: RLLib · Reinforcement learning · Gradient temporal-difference

1 Overview

The RoboCup initiative presents a real-time, dynamic, complex, adversarial, and stochastic multi-agent environments for agents to learn and reason about different problems [10]. Reinforcement Learning (RL) is a framework that provides a set of tools to design sophisticated and hard-to-engineer behaviors [11]. RL agents naturally formalize their learning goals in two layers: (1) physical layers – where controlling or predicting of functions relate to sensorimotor interactions such as walking, kicking, so forth; and (2) decision layers – with dynamically emerging behaviors through actions, options, or knowledge representations. Therefore, the RL paradigm includes controlling and prediction; it also provides means to represent highly expressive knowledge using General Value Functions (GVFs) from sensorimotor streams [14].

RLLib is a lightweight C++ template library that implements incremental, standard, and gradient temporal-difference learning algorithms in RL to effectively solve problems defined in physical and decision layers. The library is designed and written specifically for robotic applications and embedded devices such as RoboCup 3D soccer simulation agents, physical NAO V4 humanoid robots, and Tiva C series launchpad microcontrollers that operate under fast duty cycles. The library is first released on May 17, 2013 under the open source license "Apache License, Version 2.0", and the latest release version is v2.2.

© Springer International Publishing Switzerland 2015
L. Almeida et al. (Eds.): RoboCup 2015, LNAI 9513, pp. 356–364, 2015.
DOI: 10.1007/978-3-319-29339-4_30

The main objective of this application paper is to describe the design, correctness, implementation, and efficacy of RLLib across multiple hardware platforms. Henceforth, we have organized the paper as follows. Section 2 describes the existing applications. A brief description on the implemented algorithms and features are described in Sect. 3, while, the interpretations related to the algorithms are summarized in Sect. 4. The main concepts of the framework is described in Sect. 5. The experiments and evaluations are provides in the penultimate Sect. 6. Finally, we conclude the paper with a summary and extensions in Sect. 7.

2 Related Work

There exist RL development platforms published by researchers for specific RL problems and for general use. Notably, the most common RL development platform is RL-GLUE[1] with RL-LIBRARY[2] packages [29]. It provides a language-independent platform over text based massage passing among agents and environments. RL-GLUE is being used in RL competitions in ICML and NIPS workshops. PyBrain[3] [19] and RLPy[4] are libraries written in Python to formulate and learn from RL problems. RL toolbox[5], libpgrl[6], YORLL[7], and rllib[8] [8] are C++ based platforms to develop RL algorithms in different scenarios, while CLSquare[9] [9] is a standardized platform for testing RL problems with on-policy batch controllers. BURLAP[10] [7], PIQLE[11] [6], MMF[12], QCON[13], and RLPark[14] are Java platforms that model and learn from RL problems. MDP Toolbox[15] is an Octave based RL development platform. dotRL[16] [18] is a .NET platform for rapid RL method development and validation.

RLLib is significantly differs from the existing platforms because of the following: (1) the library is written and designed specifically for applications and devices where limited computational resources are available. Therefore, the memory footprint as well as the computational requirements that are needed by the library have been optimized; (2) a configurable C++ template functions to

[1] http://glue.rl-community.org.
[2] http://library.rl-community.org.
[3] http://pyrain.org/.
[4] http://acl.mit.edu/RLPy/.
[5] http://www.igi.tu-graz.ac.at/gerhard/ril-toolbox.
[6] https://code.google.com/p/libpgrl/.
[7] http://www.cs.york.ac.uk/rl/software.php.
[8] http://malis.metz.supelec.fr/spip.php?article122.
[9] http://ml.informatik.uni-freiburg.de/research/clsquare.
[10] http://burlap.cs.brown.edu/.
[11] http://piqle.sourceforge.net/.
[12] http://mmlf.sourceforge.net/.
[13] http://sourceforge.et/p/elsy/wiki/Home/.
[14] http://rlpark.github.io/.
[15] http://www7.inra.fr/mia/T/MDPtoolbox/.
[16] http://sourceforge.net/projects/dotrl/.

synchronize with application or device hardware requirements; (3) the library emphasizes more on learnable knowledge representation and reasoning from sensorimotor streams; (4) a clean and transparent API exists that enables users to model their RL problems easily; and (5) a self-contained C++ template library covers plethora of incremental, standard, and gradient temporal-difference learning algorithms in RL that is published to-date (e.g., [22,28]). Our library has been successfully used in [1] to learn role assignment in RoboCup 3D soccer simulation agents.

3 Features

RLLib implements and features: (1) off-policy prediction algorithms: (GTD(λ) and GQ(λ)) [14]; (2) off-policy control algorithms: Greedy-GQ(λ) [14] and (Softmax GQ(λ) and Off-PAC) [5]; (3) on-policy algorithms: (TD(λ), SARSA(λ), Expected SARSA(λ), and Actor-Critic (continuous and discrete actions, discounted, averaged reward settings, so forth)) [25], (Alpha Bound TD(λ) and SARSA(λ)) [3], and (True TD(λ) and SARSA(λ)) [21]; (4) incremental supervised learning algorithms: Adaline [2], (IDBD and Semi-Linear IDBD) [26], and Auto-Step [17]; (5) discrete and continuous policies: (Random, Random X percent bias, Greedy, ϵ-greedy, Boltzmann, Normal, and Softmax); (6) sparse feature extractors (e.g., Tile Coding) with pluggable hash functions [25]; (7) an efficient implementation of the dot product for sparse coder based feature representations; (8) benchmark environments: (Mountain Car, Mountain Car 3D, Swinging Pendulum, Helicopter, and Continuous Grid World) [25]; (9) optimization for very fast duty cycles (e.g., using culling traces, RLLib is tested on the RoboCup 3D simulator agents, physical NAO V4 humanoid robots, and Tiva C series launchpad microcontrollers); (10) a framework to design complex behaviors, predictors, controllers, and represent highly expressive learnable knowledge representations in RL using GVFs; (11) a framework to visualize benchmark problems; and (12) a plethora of examples demonstrating on-policy and off-policy control experiments.

4 Prelude

The standard RL models an AI agent and its environment interactions in discrete time steps $t = 1, 2, 3, \ldots$. The agent senses the state of the world at each time step $S_t \in \mathcal{S}$ and selects an action $A_t \in \mathcal{A}$. One time step later the agent receives a scalar reward $R_{t+1} \in \mathbb{R}$, and senses the state $S_{t+1} \in \mathcal{S}$. The rewards are generated according to a reward function $r : S_{t+1} \rightarrow \mathbb{R}$. The objective of the RL is to learn a stochastic action-selection policy $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$, that gives the probability of selecting each action in each state, such that the agent maximizes rewards summed over the time steps [25].

An interpretation assigns semantics to a set of entities in a domain of discourse. In RL, a set of arbitrary value functions contains the entities over which the interpretation is defined [14–16,27]. In this context, knowledge is represented

as a large number of approximate value functions each with its: (1) target policy (π); (2) pseudo-reward function ($r : \mathcal{S} \rightarrow \mathbb{R}$); (3) pseudo-termination function ($\gamma : \mathcal{S} \rightarrow [0, 1]$); and (4) pseudo-terminal-reward function ($z : \mathcal{S} \rightarrow \mathbb{R}$).

The GVFs are defined over the four functions: $\pi, \gamma, r,$ and z. However, γ function is more substantive than reward functions as the termination interrupts the normal flow of state transitions. In pseudo mode, the standard termination is omitted. As an example, in robotic soccer, a base problem can be defined as duration in which a goal is scored by either teams. We can consider a pseudo-termination has occurred when a striker is changed. A GVF with respect to a state-action function is defined as $q^{\pi,\gamma,r,z}(s, a)$. Therefore, these four functions provide the meaning to the GVF in the form of a question.

In continuous state and action spaces, a value function is represented using a function approximator, $\hat{q}$, which amounts to a majority of problems encountered in practice. As an example, in linear function approximation, there exists a weight vector, $\boldsymbol{\theta} \in \mathbb{R}^N$, to be learned. Hence, an approximate GVF is defined as: $\hat{q}(s, a, \theta) = \boldsymbol{\theta}^{\mathrm{T}} \boldsymbol{\phi}(s, a)$, such that, $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^N \rightarrow \mathbb{R}$. The weights are learned using on/off-policy algorithms implemented in Sect. 3. The approximate value function, $\hat{q}$, is the answer to the question formed by its value function, q. Therefore, the truth value of q is measured by $\hat{q}$, which in return completes the intended interpretation.

5 Platform

RLLib closely follows the design principles and recommendations presented in [13,24]. The development of the library has taken significant efforts to minimize memory footprint as well as computational requirements that are requested by RL problems. Since RLLib specifically emphasizes on learnable knowledge representation and reasoning, it has been modularized based on on-policy and off-policy RL methods. In addition, RLLib provides implementation of incremental supervised learning algorithms that can be used simultaneously with RL problems.

Listings 1 and 2 provide the minimal pseudo-code to setup on/off-policy RL agents. The controlling, behavior learning, and learnable knowledge representation problems should use `"ControlAlgorithm.h"` header file. The algorithms related to predictions and supervised learning problems are implemented in `"PredictorAlgorithm.h"` and `"SupervisedAlgorithm.h"` header files. All C++ templates implemented in the library are under the namespace `RLLib`, and use a single parameter `T`. This parameter could be a C++ primitive type as shown in Listings 1 and 2 or a complex object defined by the user. It is our experience that majority of RL problems can be defined using a primitive type. Devices such as Tiva C launchpad microcontrollers supports single precision floating point representations. Therefore, the templates should be initialized with `float` primitive type.

```
// ----------------------------------------------
 1 include "ControlAlgorithm.h"
 2 include "RL.h"
 3 using namespace RLLib;
// RL Problem ------------------------------------
 4 RLProblem<double>*  problem = ...;
// Projector -------------------------------------
 5 Hashing<double>* hashing = ...;
 6 Projector<double>*  projector = ...;
 7 StateToStateAction<double>*  toStateAction = ...;
// Predictor -------------------------------------
 8 Trace<double>* e = ...;
 9 GQ<double>* gq = ...;
// Policies π and π_b ----------------------------
10 Policy<double>*  target = ...;
11 Policy<double>*  behavior =...;
// ----------------------------------------------
12
13
// Controller ------------------------------------
14 OffPolicyControlLearner<double>*  control = ...;
// Runner ----------------------------------------
15 RLAgent<double>*  agent = ...;
16 Simulator<double>*  sim = ...;
17 sim->run(); // OR sim->step();
// ----------------------------------------------
```

Listing 1: Pseudo-code for action-value methods.

```
// ----------------------------------------------
 1 include "ControlAlgorithm.h"
 2 include "RL.h"
 3 using namespace RLLib;
// RL Problem ------------------------------------
 4 RLProblem<double>*  problem = ...;
// Projectors ------------------------------------
 5 Hashing<double>* hashing = ...;
 6 Projector<double>*  projector = ...;
 7 StateToStateAction<double>*  toStateAction = ...;
// Critic ----------------------------------------
 8 Trace<double>* critice = ...;
 9 GTDLambda<double>* critic = ...;
// Policies π and π_b ----------------------------
10 PolicyDistribution<double>*  target = ...;
11 Policy<double>* behavior =...;
// Actor -----------------------------------------
12 Traces<double>*  actoreTraces = ...;
13 ActorOffPolicy<double>*  actor = ...;
// Controller ------------------------------------
14 OffPolicyControlLearner<double>*  control = ...;
// Runner ----------------------------------------
15 RLAgent<double>*  agent = ...;
16 Simulator<double>*  sim = ...;
17 sim->run(); // OR sim->step();
// ----------------------------------------------
```

Listing 2: Pseudo-code for policy gradient methods .

In line 4, we define the RL problem using an instance of the template `RLProblem<T>`. This template as well as `RLAgent<T>` and `Simulator<T>` (lines 15–16) templates are defined in `"RL.h"` header file. An instance of the template `Projector<T>` (line 6) extracts features from the state variables. These features are part of a function approximation architecture (linear or non-linear), e.g., tile based sparse features [25, Sect. 8.3.2] or compact features [12], suitable for the problem. Some feature extractors require a hashing function, that is defined in line 5. For action-value functions, the features could also include actions (or options), that is defined in `StateToStateAction<T>` (line 7). In Listing 1, lines 8–9 define the predictor, which is used in off-policy controller (line 14), while Listing 2 defines the critic that is used in the actor-critic controller. Lines 12–13 in Listing 2 define the actor for the prior controller. Lines 10–11 define the target policy (the smooth policy distributions in Listing 2) and the behavior policy. Line 15 defines the RL agent that is used in the simulator (line 16) simultaneously with the RL problem.

In simulations (e.g., [23]), specifically when the simulator has the control over the perception-actuation cycles, the runner is executed with `run()` in line 17. In practical problems (e.g., [20]), where the agents and the environments run on disjoint processes, the runner will wait for the percepts, then updates the agent, which in return transmits the actions to the environment. In such situations, the runner is executed with `step()` in line 17. It is to be noted that either in simulations or in practical problems the runner will execute the same update steps, such that, the user will experience the same set of execution steps. A practitioner can construct the C++ objects in lines 4–16 in an initialization subroutine, and execute line 17 in a subroutine that calls in every duty cycle.

There are complex combination of RL algorithms used in practice. RLLib allows many combination of these algorithms by changing a few lines of code in Listings 1 and 2. For example, in oder to implement on-policy action-value methods, a practitioner can change the predictor in line 9 to `Sarsa<T>` and

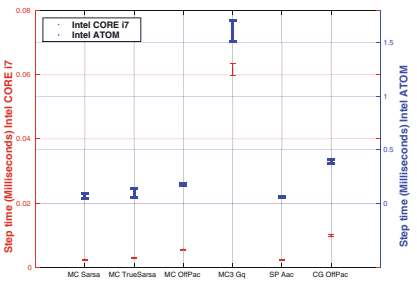

Fig. 1. Step update times in milliseconds. The thick error bars (blue) show the step time for Intel ATOM, while the thin error bars (red) show the step update time for Intel CORE–i7 (Color figure online).

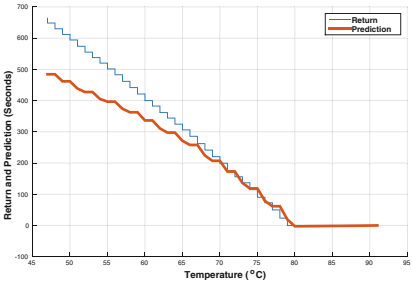

Fig. 2. Predicting the time to shutdown in seconds. The bold line (red) shows the prediction to shutdown from a given temperature of NAO left knee. The thin line (blue) shows the actual return (Color figure online).

the controller in line 14 to `OnPolicyControlLearner<T>` in Listing 1. Similarly, different combination of RL algorithms can be included in Listing 2 for actor-critic, and parameterized policies.

6 Experiments

This section provides experiments: (1) to validate the effectiveness of the library across multiple hardware platforms; and (2) the ability to answer a subjectively posed predictive question using the conceptualization of GVFs.

Figure 1 shows the step time in milliseconds, i.e., the time an algorithm requires to update its parameters from the observations, for a set of benchmark problems popular in RL literature. We have considered two hardware platforms: (1) Intel CORE–i7 2.2GHz laptop; and (2) Intel ATOM 1.6GHz CPU available on NAO humanoid robot. MC Sarsa, MC TrueSarsa, and MC OffPac represent the two dimensional mountain car benchmark problem solved using sarsa, true sarsa, and off-policy actor-critic algorithms. The reader is referred to [5,21,23] that describe the problems, feature extractions, and parameter settings respectively. MC3 Gq describes the three dimensional mountain car [30] solved using greedy-GQ algorithm [14]. SP Aac and CG OffPac represent swinging pendulum and continuous grid-world problems solved using average [4] and off-policy [5] actor-critic algorithms. Even though the step update on NAO platform is on average 30.64 times slower than the laptop, the step update is suitable for real-time operations (worst case time is approximately 1.5 ms).

Our second experiment poses a question of the form "How much time remaining on NAO before the left knee temperature reaches above 80 °C?". We have configured a NAO robot to walk in the standard platform league soccer field. We have started the robot in resting temperature, and stopped the experiment when at least one of the joints reached the critical temperature (>80 °C). We have used GQ(λ) [5] with the GVF question encoding: π(s,a) = 1, $r(s) = 0.01, z(s) = 0$ seconds (we have queried the robot sensor values at 100 Hz), $\forall s \in \mathcal{S}$, and $\gamma(s) = 0$

if the left knee temperature is greater than 80 °C, otherwise $\gamma(s) = 1$. We have used the answer encoding: $\lambda(s) = 0.4$, $\forall s \in \mathcal{S}$ and single tiling with 28 regions of the effective temperature range (47 °C, 91 °C) with 512 memory (the size of the feature, eligibility, and primary and secondary parameter vectors). We set the two step size parameters to $\alpha_v = 0.2$ and $\alpha_w = 0.000001$. As shown in Fig. 2, the agent has learned accurate predictions to shutdown from returns.

7 Conclusion

RLLib framework and its features, as presented in Sects. 1 and 5, have been fully implemented, empirically tested, and released to public from project website at: http://web.cs.miami.edu/home/saminda/rllib.html. In addition, RLLib provides testbeds, intuitive visualization tools, and extension points for complex combination of RL algorithms, agents, and environments. Compared to existing platforms, RLLib has been successfully deployed in different hardware configurations due to its low memory footprint and computational efficiency. The library is also compatible with devices support by Energia, an open-source electronics prototyping platform.

References

1. Abeyruwan, S., Seekircher, A., Visser, U.: Dynamic role assignment using general value functions. In: AAMAS 2013, Adaptive Learning Agents Workshop (2013)
2. Bishop, C.M.: Pattern Recognition and Machine Learning. Information Science and Statistics, 1 edn. Springer, Heidelberg (2007)
3. Dabney, W., Barto, A.G.: Adaptive step-size for online temporal difference learning. In: AAAI Conference on Artificial Intelligence (2012)
4. Degris, T., Pilarski, P.M., Sutton, R.S.: Model-free reinforcement learning with continuous action in practice. In: American Control Conference (ACC), pp. 2177–2182. IEEE (2012)
5. Degris, T., White, M., Sutton, R.S.: Off-policy actor-critic. In: Proceedings of the 29th International Conference on Machine Learning (ICML), pp. 457–464 (2012)
6. Delepoulle, F.D.C.S.: PIQLE: a platform for implementation of q-learning experiments. In: Neural Information Processing Systems (NIPS), Workshop on Reinforcement Learning Benchmarks and Bake-off II (2005)
7. Diuk, C., Cohen, A., Littman, M.L.: An object-oriented representation for efficient reinforcement learning. In: Proceedings of the 25th International Conference on Machine Learning (ICML), pp. 240–247 (2008)
8. Frezza-Buet, H., Geist, M.: A C++ template-based reinforcement learning library: fitting the code to the mathematics. J. Mach. Learn. Res. (JMLR) **14**(1), 625–628 (2013)
9. Hafner, R., Riedmiller, M.: Case study: control of a real world system in CLSquare. In: Proceedings of the NIPS Workshop on Reinforcement Learning Comparisons, Whistler, British Columbia, Canada (2005)
10. Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawai, E., Matsubara, H.: RoboCup: a challenge problem for AI and robotics. In: Kitano, H. (ed.) RoboCup 1997. LNCS, vol. 1395, pp. 1–19. Springer, Heidelberg (1998)

11. Kober, J., Peters, J.: Reinforcement learning in robotics: a survey. In: Wiering, M., van Otterlo, M. (eds.) Reinforcement Learning. ALO, vol. 12, pp. 579–610. Springer, Heidelberg (2012)
12. Konidaris, G., Osentoski, S., Thomas, P.: Value function approximation in reinforcement learning using the Fourier basis. In: Proceedings of the 25th Conference on Artificial Intelligence, pp. 380–385 (2011)
13. Kovacs, T., Egginton, R.: On the analysis and design of software for reinforcement learning, with a survey of existing systems. Mach. Learn. **84**(1–2), 7–49 (2011)
14. Maei, H.R.: Gradient temporal-difference learning algorithms. Ph.D. thesis, University of Alberta (2011)
15. Maei, H.R., Sutton, R.S.: GQ(λ): a general gradient algorithm for temporal-difference prediction learning with eligibility traces. In: Proceedings of the 3rd Conference on Artificial General Intelligence (AGI), pp. 1–6. Atlantis Press (2010)
16. Maei, H.R., Szepesvári, C., Bhatnagar, S., Sutton, R.S.: Toward off-policy learning control with function approximation. In: Proceedings of the 27th International Conference on Machine Learning (ICML), pp. 719–726 (2010)
17. Mahmood, A.R., Sutton, R.S., Degris, T., Pilarski, P.M.: Tuning-free step-size adaptation. In: Acoustics, Speech and Signal Processing (ICASSP), pp. 2121–2124. IEEE (2012)
18. Papis, B., Wawrzynski, P.: dotRL: a platform for rapid reinforcement learning methods development and validation. In: 2013 Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 129–136 (2013)
19. Schaul, T., Bayer, J., Wierstra, D., Sun, Y., Felder, M., Sehnke, F., Rückstieß, T., Schmidhuber, J.: PyBrain. J. Mach. Learn. Res. **11**, 743–746 (2010)
20. Seekircher, A., Abeyruwan, S., Visser, U.: Accurate ball tracking with extended Kalman filters as a prerequisite for a high-level behavior with reinforcement learning. In: The 6th Workshop on Humanoid Soccer Robots at Humanoid Conference, Bled (Slovenia) (2011)
21. Seijen, H.V., Sutton, R.: True online TD(λ). In: Jebara, T., Xing, E.P. (eds.) Proceedings of the 31st International Conference on Machine Learning (ICML). JMLR Workshop and Conference Proceedings, pp. 692–700 (2014)
22. Sigaud, O., Buffet, O.: Markov Decision Processes in Artificial Intelligence. Wiley, New York (2013)
23. Sutton, R.S.: Generalization in reinforcement learning: successful examples using sparse coarse coding. In: Advances in Neural Information Processing Systems 8, pp. 1038–1044. MIT Press (1996)
24. Sutton, R.S.: a standard interface for reinforcement learning software in C++. http://webdocs.cs.ualberta.ca/sutton/RLinterface/RLI-Cplusplus.html. Accessed 12 July 2015
25. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998)
26. Sutton, R.S., Koop, A., Silver, D.: On the role of tracking in stationary environments. In: Proceedings of the 24th International Conference on Machine Learning, pp. 871–878. ACM (2007)
27. Sutton, R.S., Modayil, J., Delp, M., Degris, T., Pilarski, P.M., White, A., Precup, D.: Horde: a scalable real-time architecture for learning knowledge from unsupervised sensorimotor interaction. In: Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pp. 761–768 (2011)

28. Szepesvári, C.: Algorithms for Reinforcement Learning. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, San Rafael (2010)
29. Tanner, B., White, A.: RL-Glue: language-independent software for reinforcement-learning experiments. J. Mach. Learn. Res. **10**, 2133–2136 (2009)
30. Taylor, M.E., Kuhlmann, G., Stone, P.: Autonomous transfer for reinforcement learning. In: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), vol. 1, pp. 283–290 (2008)