dotRL: A platform for rapid Reinforcement Learning methods development and validation

Bartosz Papis, Paweł Wawrzyński
Institute of Control and Computation Engineering, Warsaw University of Technology

Abstract—This paper introduces dotRL, a platform that enables fast implementation and testing of Reinforcement Learning algorithms against diverse environments. dotRL has been written under .NET framework and its main characteristics include: (i) adding a new learning algorithm or environment to the platform only requires implementing a simple interface, from then on it is ready to be coupled with other environments and algorithms, (ii) a set of tools is included that aid running and reporting experiments, (iii) a set of benchmark environments is included, with as demanding as Octopus-Arm and Half-Cheetah, (iv) the platform is available for instantaneous download, compilation, and execution, without libraries from different sources.

Index Terms—Reinforcement learning, evaluation platform, software engineering

I. INTRODUCTION

In the area of Reinforcement Learning (RL) [1] algorithms are developed that learn reactive policies for sequential decision making and control. Research in RL is based on the paradigm of micro–worlds: ideas are tested and demonstrated with the use of decision-making and control problems that can be defined analytically and reimplemented by others. This has forced researchers to spend a lot of time developing their experimental platforms. In order to help others and enable fair comparison of the ideas, many researchers have published their platforms: RL-Glue [2], PyBrain [3], CLSquare [4], RLT [5], PIQLE [6], lipqrl [7], MDP Toolkit [8], MMLF [9], or QCON [10]. The general design principles for RL platforms were analysed in [11].

The purpose of this paper is to introduce another platform, dotRL, for development of RL algorithms. Although the platforms reduce the space for yet another project, it also demonstrates that a researcher developing a new idea in RL or a student getting familiar with this field still prefers writing their own platform from scratch instead of using an existing one. That is why the main principle that we adopted when designing our platform was as follows: the user should spend as little time as possible installing, getting familiar with the platform, and writing code, before they are ready to run their own agent or environment.

A. Related work

Perhaps the oldest and best-known RL platform is RL-Glue [2]. It dates back to 1996 through a project by Rich Sutton and Juan Carlos Santamaria called RL-Interface. RL-Glue has been a protocol specified by annual RL competition workshops held at ICML and NIPS. RL-Glue is basically a text communication protocol over sockets, between agents and environments. Reinforcement learning toolbox (RLT) [5] is a flexible platform for development learning algorithm in various scenarios: MDP, POMDP, and imitation learning. The price of this flexibility is the complexity of this platform and difficulty of its use. Libqrl [7] focuses on planning and reinforcement learning in a distributed environment. Maja machine learning framework (MMLF) [9] supports not only RL but also model-based learning and direct policy search. It enables automated experimentation with the use of XML configuration files. PyBrain [3] is a general machine learning library, that also includes RL, but focuses on neural networks. Object-oriented platforms written in Java include PIQLE [6], RL-Park [12], and Teachingbox [13]. Another platform, YORLL [14], is written in C++.

B. Requirements and basic assumptions

The dotRL platform is designed to minimize the time spent by its user on technical and infrastructural details. The user should focus almost all of their effort on dealing with purely scientific issues. In order to meet this requirement, the design of dotRL is based on the following assumptions and characteristics:

1) Altogether, dotRL is a solution written under .NET 4.5 framework, Windows operating system, and Visual Studio 2010. As a result, further development of dotRL may be based on all the tools provided with Visual Studio and .NET technology.

2) Having been downloaded and opened with Visual Studio, it is ready to be compiled and run.

3) In order to add a new agent or a new environment to the platform, one only needs to implement a class with an appropriate interface. After compilation, the platform alone is able to couple this new entity to other environments or agents.

4) Each agent and environment is designed for one particular problem type. The problem type defines the types of state and action spaces. They may be continuous (i.e., contain vectors of reals), discrete (contain vectors of integers), and possibly others.

5) A set of tools is provided with dotRL that enables launching many learning runs with the same setting and getting logs almost directly insertable to a scientific paper. Tools for implementing agents, such as neural networks, are also included.

6) A set of exemplary agents and environments are provided with the platform. Those include as complex en-
environments as Octopus-Arm [15], [16] and Half-Cheetah [17], [18].

7) The platform is fully compatible with RL-Glue [2].

To our knowledge, the platform presented in this paper is the first full-featured platform written under .NET, and the first one in which adding a new agent or environment only requires implementing a single class. Especially this last feature is helpful in rapid development and validation of new algorithms.

The aforementioned notion of problem types is based on the following observation: An agent is usually applicable, without modification only to environments with compatible state and action space types. No one really implements a learning algorithm that, in the same form, is applicable to several problem types. It is possible to do so, but almost always means that the agent will do something completely different for different versions of environment it deals with at the moment.

Additional contribution of this work is RL-Glue codec for .NET platform.

dotRL is an open source software under BSD license and hosted on sourceforge.net [19]. We welcome anyone to contribute to the project.

C. Organization of the paper

The remaining part of the paper is organized as follows. Sec. II presents an overview of the user interface, sec. III defines basic modules and components of the dotRL platform. Another subsection presents the interaction protocol between an agent and an environment that the platform supports. Sec. IV explains how to use a new component (agent or environment) with the platform. Sec. V elaborates on integration of dotRL with the RL-Glue protocol. Sec. VI concludes the paper and indicates directions of future development of the platform.

II. USER INTERFACE

Typical usage scenario of the dotRL solution, when the user wants to test an existing agent on an existing environment consists of the following steps:

1) Click the “Experiment” menu item from the “New” menu,
2) Choose an environment from the list of available environments,
3) Choose an agent from the list of available agents compatible with the chosen environment,
4) Configure parameters of the chosen environment and agent
5) Configure reporting parameters,
6) Configure experiment parameters (i.e. number of episodes, maximum number of steps in one episode),
7) Click “OK” when finished configuring the experiment,
8) Click “Background learning” or “Real time learning”,
9) Click “Present policy” and/or view the created report file.

The user can modify parameters of the ongoing experiment. Details on extending the platform’s set of components (agents or environments) are provided in Section IV. An example view of the application during configuration of an experiment is presented in Figure 1. Running experiment is presented in Figure 3, and a screen presenting functionality allowing more than one simultaneous experiments to run is shown in Figure 4.

To configure the reporting parameters “Add report file” button in “Reporting” tab needs to be clicked. Then, the user can either choose to use simple configuration and choose from the standard set of report columns, or to configure their own report:

1) For each report file tab:
   a) Choose one of report triggers
   b) Click “Add new column” for each desired column in the output file
   c) Choose one of available data sources and a way to accumulate their values

ReportTrigger and DataSource objects are explained in detail in Section III-C. An example view of the application during reporting configuration is presented in Figure 2.

For interacting with RL-Glue one of these two actions must be taken:

- Choose the RGLueAgent or RGLueEnvironment in the component choice window after choosing to create new experiment
- Start an RL-Glue experiment to connect to RL-Glue core.

Integration with RL-Glue components is explained in detail in Section V.
III. DOTRL COMPONENTS

This section presents the domain model [20] of the dotRL solution. Section III-A presents the set of core entities which reflect key notions of an RL experiment. Section III-B presents how these components interact with each other during an experiment.

A. Classes

Learning algorithms, called Agents in RL are represented as subclasses of the Agent< TStateSpaceType, TActionSpaceType > base class. Problems to solve by these algorithms, called Environments in RL are represented as subclasses of the Environment< TStateSpaceType, TActionSpaceType > base class. Environments can have continuous or discrete state transition function and they accept real or integer vectors as actions. This divides them into four groups, three of which are commonly addressed, and which we call problem types: continuous state & continuous action, continuous state & discrete action, discrete state & discrete action. Each agent and environment is dedicated to one problem type and this is made explicit in dotRL in the form of generic parameters of Agent and Environment base classes. Interaction between an agent and an environment is called Experiment. This whole design is modelled with classes presented in Figure 5.

Agent represents the class hierarchy of all agents implemented in dotRL, with Agent< TStateSpaceType, TActionSpaceType > (in Figure 5 generic arguments are omitted for clarity) being their base class. Agent’s responsibility is to decide which Action to take in given Environment’s State, and to improve its policy with received Samples. Details on how to implement an agent are provided in Section IV.

Similarly, Environment represents the hierarchy of classes which represent RL problems to be solved by the Agents. The class Environment< TStateSpaceType, TActionSpaceType > (in Figure 5 generic arguments are omitted for clarity) is the base class for any environment implemented in dotRL. Environment’s responsibility is to simulate a designed behaviour, reacting to given Actions by changing it’s State and providing a Reinforcement. Unlike some other solutions (like PyBrain [3]) we do not divide responsibility of modelling a behavior and assigning reinforcement between two separate objects. Theoretically, it would lead to a more accurate domain model and it is a valuable idea, but it makes development more time-consuming and this opposes our requirements. Different rewarding policies can be easily implemented using environment’s parameters.

The Experiment models a key notion in RL research — an experiment, i.e. a continuous interaction between an Agent and an Environment. Experiment’s responsibilities are: controlling the course of an experiment (i.e. informing about beginning and ending of an episode, evaluating finish conditions) and passing information between an agent and an environment (States, Actions, Samples and Reinforcements), and passing information to classes responsible for reporting functionality.

State< TStateSpaceType >, Action< TActionSpaceType > and Reinforcement (again, generic parameters omitted for clarity in Figure 5) are simple wrapper classes for vectors and numbers to make RL domain notions explicit in the code — they are not essential, but they make the design clear and explicit, and improve implementation’s readability.

EnvironmentDescription< TStateSpaceType, TActionSpaceType > (again, generic parameters omitted for clarity in Figure 5) is a class containing information about the structure of an environment. The details about its contents are provided in Section IV.

Presentation class provides a root for hierarchy of classes that are used to visualize the state of the environment. Its responsibility is to draw a visualization of a given state on a given canvas object (.NET’s System.Drawing.Graphics). It is used only when user chooses “Policy presentation” mode in the user interface.

Sample represents a smallest piece of information in a RL experiment. Sample consists of:

- PreviousState: a state in which the Environment was.
- Action: an action taken by the Agent for state PreviousState.
- CurrentState: a resulting state after taking action Action in state PreviousState.
- Reinforcement: a reinforcement received after taking action Action in state PreviousState.

The use of samples allows the implementation of an agent to be stateless — no information needs to be stored between calls to various agent’s methods (such as EpisodeStarted, GetAction, etc.). More details are available in Section IV.
B. Operation sequence

We propose to adopt a simple interaction scenario based on explicit interfaces. Many existing RL platforms use typical setting in which subsequent method calls (episode start, step, episode end) implicitly rely on each other, forcing agent’s implementation to be a state machine. This is not always the most convenient way, and such interface does not follow readable code guidelines [20]. The proposed sequence of method calls between components during an experiment is presented in Figure 6.

After the user initiates a new experiment instances of chosen classes are being automatically created: a subclass of the Agent< TStateSpaceType, TActionSpaceType > base class and a subclass of the Environment< TStateSpaceType, TActionSpaceType > base class (generic arguments are omitted for clarity in Figure 6). First, the user configures the parameters of the experiment (i.e. number of episodes, number of steps in each episode), agent and environment. Then, after experiment passes the information about the environment to the agent, a loop common to all RL experiments is being started. Each episode consists of a sequence of repeatedly executed steps:

1) The current state of the environment is retrieved by calling the GetCurrentState method.
2) If the current state is terminal or the current episode should end because of its duration limit, agent’s EpisodeEnded method is called, and a new episode is started by calling StartEpisode environment’s method and EpisodeStarted agent’s method.
3) Agent’s action for current state is retrieved with call to the GetActionWhenLearning method.
4) The Environment is informed about the consequences of executed action via call to the Learn method.
5) The reinforcement and the new current state are retrieved from the Environment as return values from the PerformAction and GetCurrentState methods.
6) The Agent is informed about the consequences of executed action via call to the Learn method.

If the user wishes only to see how the agent behaves without changing it’s policy they can choose “Policy presentation” mode. In this mode, another copy of the environment is used and agent’s GetActionWhenNotLearning method is used (opposed to GetActionWhenLearning) so there is no interference with the learning process (provided that the implementation of GetActionWhenNotLearning is correct and truly does not influence the learning process).

C. Reporting

A valuable functionality of dotRL is provided by the reporting mechanisms. When configuring an experiment the user can setup multiple output log files. This is done through three useful notions: ReportTrigger, DataSource and DataAccumulator. A report trigger is a class that decides when to write a line to the output file. Lots of report triggers have already been implemented, such as: TotalStepCountReportTrigger which causes emitting a log file line at configured intervals of steps in an experiment, EpisodeCountReportTrigger which causes emitting a log file line at configured intervals of episodes in the whole experiment.

DataSource consists of an object and it’s field, which’s value
will be reported to the output file. There are three main objects which provide data: Experiment, Agent and Environment. The experiment provides typical experiment information, like the number of steps executed so far, or the reinforcement received by the agent. Data exposed by the agent and the environment depends on the creator of these components. Any useful piece of data can be accessed by the reporting functionality as easy, as marking any component’s field with ReportedValue attribute. An example is provided in QLearningAgent which exposes it’s td field, containing recently computed temporal-difference value.

DataAccumulator allow some simple manipulations on the data read from data sources. The most common are no-op data accumulator (CurrentValueDataAccumulator) which just outputs the returned value and AverageSinceTriggerDataAccumulator which accumulates the data between each report file line and calculates average.

D. Implemented components

Currently, the following components are implemented in dotRL:

- Environments:
  - Cart-Pole Swing Up [21]
  - Double Inverted Pendulum on a Cart [22]
  - Acrobot [23]
  - Robot weightlifting [24]
  - Kimura’s Robot [25]
  - Half Cheetah [17]
  - Octopus Arm [16]
  - Coffee task [26]
  - Grid
- Agents:
  - Actor-Critic [27]
  - Actor-Critic with Experience Replay [17]
  - Q-Learning [28]
  - SARSA [29]

IV. ADDING NEW COMPONENTS

We focus our design to make adding new agents and environments as simple as possible. This allows a researcher to spend most of his time on substantial work instead of dealing with technical details. Developing a new agent or a new environment is most straightforward: one needs just to subclass the Agent<TStateSpaceType, TActionSpaceType> class or the Environment<TStateSpaceType, TActionSpaceType> class, respectively. The TStateSpaceType and TActionSpaceType generic arguments need to be set to types corresponding to desired problem type (for example: setting TStateSpaceType = double, TActionSpaceType = int allows creation of a continuous state & discrete action agent/environment).

Each component, once implemented, will appear automatically in the user interface. If additionally a subclass of the Presenter class is supplied, the environment’s state will be visualized in the experiment’s window. Otherwise the default presenter will be used, which just prints raw state and reinforcement information. The implementation of Experiment<TStateSpaceType, TActionSpaceType> (green component in Figure 5) is provided by the dotRL platform, and is fully configurable through the user interface.

Another convenience is automatic handling of component’s parameters. Every Agent or Environment can have any of its fields or properties (doesn’t matter whether private, protected, public, static or instance related) marked with one of Parameter or NumericParameter attributes. Such fields will appear in a configuration dialog window before starting each experiment, allowing the user to tune the component’s behavior. Also, if any component uses another component (for example one wants to implement an environment similar to an existing one, and reuses the latter as a part of the new one) its parameters will be also handled automatically.

A. Adding a new environment

Subclassing the Environment<TStateSpaceType, TActionSpaceType> class requires implementing the following methods (for clarity, generic arguments have been omitted):

- EnvironmentDescription GetEnvironmentDescription(): called to retrieve information about the environment
- void StartEpisode(): called when a new episode begins
- Reinforcement PerformAction(Action action): called to execute action and retrieve reinforcement

The first method is called to transfer information about the structure of the problem to the agent. Usually agents require information about the problem’s state, action and reinforcement spaces. Such information is stored in EnvironmentDescription<TStateSpaceType, TActionSpaceType> class, which has two instances of SpaceDescription<TSpaceType> classes (one for state space and one for action space) and one instance of DimensionDescription<TSpaceType> class for describing the reinforcement space. SpaceDescription<TSpaceType> consists of DimensionDescription<TSpaceType> instance for each described dimension. DimensionDescription<TSpaceType> contains: minimum value, maximum value, average value and standard deviation.

Not all of these fields are always used. Typically, state space information contains:

- Minimum value for each state variable.
- Maximum value for each state variable.
- Average value for each state variable.
- StandardDeviation of each state variable.

Action space information:

- Minimum value for each action dimension.
- Maximum value for each action dimension.

Information about the reinforcement:

- Minimum reinforcement value.
- Maximum reinforcement value.
Despite the typical setting, all values are optional but the environment should provide as much information as possible, to allow cooperation with agents that need it.

Typical behavior of the `StartEpisode` method is to initialize environment’s state (to some predefined state, probably dependent on parameters or to a random state).

The last method, `PerformAction` is typical to RL environment implementations: it usually performs a simulation step, evaluating the consequences of the given action (calculating environment’s next state) and returns a reinforcement associated with this action in its current state.

Technically, these methods should be implemented in the paradigm of a `stateful` protocol — environment should keep track of its current state. To facilitate this and for efficiency, the `Environment<TStateSpaceType, TActionSpaceType>` base classes exposes a protected mutable `CurrentState` property. As long as it is used by `StartEpisode` and `PerformAction` methods one needs not to bother about implementing the `GetCurrentState` method.

Additionally these methods can optionally be overriden:

- `State GetCurrentState();` called to retrieve environment’s current state
- `void ParametersChanged();` called after user changes environment’s parameters
- `void ExperimentEnded();` called after the user closes the experiment window

The default implementation of the first method returns the `CurrentState` property as an immutable object. The default implementations of the two remaining methods do nothing.

Environment class must contain a parameterless constructor.

B. Adding a new agent

Subclassing the `Agent<TStateSpaceType, TActionSpaceType>` class requires implementing the following methods (for clarity, generic arguments have been omitted):

- `void ExperimentStarted(EnvironmentDescription environmentDescription);` called to pass the information about the environment to the agent
- `Action GetActionWhenNotLearning;` called to retrieve agent’s decision about an action to take in the given state, when presenting current policy
- `Action GetActionWhenLearning(State state);` called to retrieve agent’s decision about an action to take in the given state, during learning
- `void Learn(Sample sample);` called to inform the agent about a state, action that took place and the resulting next state and reinforcement

The first method is called before the start of the experiment, so the agent could prepare its internal structures accordingly to the structure of the environment (e.g., dimensions of the state, and action spaces).

The `GetActionWhenNotLearning` should return the action according to agent’s current policy for the given state, not affected by agent’s exploration policy, and in way not to interfere with agent’s internal state related to learning.

The third method, `GetActionWhenLearning` should return the action according to current agent’s policy for the given state which can be distorted for exploration. Also in this method agent can calculate or remember some additional quantities which will be needed in the `Learn` method. It is guaranteed that a call to `Learn` method will always follow a previous call to `GetActionWhenLearning`.

The `Learn` method, typical to RL agent implementations, is used to transfer experience to the agent. Agent can, for example accumulate this experience, or improve its policy at once.

Technically these methods should be implemented in the paradigm of a `stateless` protocol. Subsequent calls to `GetActionWhenLearning` or `Learn` should not be explicitly dependent. Of course, there should be an implicit dependency between these calls and calls to the `Learn` method through the agent’s policy and some internal variables used for learning. Also, the `action` returned by the `GetActionWhenLearning` method will be present in a sample given to a subsequent `Learn` call.

The `GetActionWhenNotLearning` method should act in a completely transparent way — no assumptions should be made about the moment of its execution, as user can switch to “Policy presentation” mode at any time. The environment will remain unaffected, as in “Policy presentation” mode its copy is being used.

Similarly to environments’ base class, the base class for agents exposes a mutable protected `Action` property to be modified in place for efficiency, and returned from `GetAction` as an immutable version.

Additionally these methods can optionally be overriden:

- `void EpisodeStarted(State state);` called when episode starts
- `void EpisodeEnded();` called when episode ends
- `void ExperimentEnded();` called after the user closes the experiment window

The default implementations of these methods do nothing. Agent class must contain a parameterless constructor.

C. Adding a new presenter

Subclassing the `Presenter` base class is optional, however it allows a researcher to evaluate environment’s behavior visually. Usually environments represent some imaginable object, or they are related to a real-world object. Having them drawn and being able to observe their dynamics makes it easier to verify their implementation, and to analyze agent’s behavior. To implement a presenter one needs to:

- subclass the `Presenter` base class and implement the `Draw` method,
- provide a constructor taking one argument of type of the environment to visualize.

The `Draw` method should draw the visualization of environment’s current state to the canvas held by the `Graphics` object, exposed by the `Presenter` base class. `Draw` implementation should use drawing functions also provided by the `Presenter`
base class, as they scale the drawing appropriately to window’s dimensions and aspect ratio. Presenter implementations should hold the reference to the visualized environment for themselves. Implementation of a presenter can be designed to visualize more than one environment — it just needs to provide one constructor for each visualized environment. This is useful when one environment is derived from another one.

V. INTEGRATION

A. Integration with RL-Glue

In RL research field there are many agents and environments available, however they are implemented in different platforms and languages. This problem is taken care of by the RL-Glue protocol [2]. dotRL supports it, to give its users access to the vast set of agents and environments already implemented and compatible with RL-Glue.

Generally, interaction between two integrated platforms (or components from different solutions) relies on one of them managing the course of an experiment and the other acting passively as one of experiment components: an agent or an environment. This gives two options of integrating dotRL with RL-Glue components:

1) dotRL manages the course of an experiment employing a RL-Glue agent or environment.
2) dotRL acts as RL-Glue component: an agent or an environment, while RL-Glue manages the course of the experiment.

dotRL allows both scenarios: the user can instantiate a component (agent or environment) alone, and configure it to connect to a RL-Glue server application, or user can open an ordinary experiment window choosing a special agent (RLGlueAgent) or environment (RLGlueEnvironment) type which act as proxies and encapsulate RL-Glue network communication details. Example of the first scenario is presented in Figures 7 and 8. Some other platforms, like Teachingbox [13] also provide integration with RL-Glue, but dotRL is the only platform known to the authors which allows both integration scenarios.

B. Integration with other applications

Integrating dotRL with other platforms, or single component applications is easy thanks to the mechanisms provided by the .NET framework. For example:

- C/C++ code, compiled to a DLL library is easily accessible from .NET through the Platform Invoke Services (P/Invoke) or it Just Works (IJW).
- Python code can be accessed with solutions like IronPython [30] which provide Python virtual machine implementation in .NET.
- Because Matlab gives access to its API through DLL interface, it also possible to run Matlab scripts using P/Invoke mechanism. Matlab’s COM interface can also be used.
- Interaction with XML based communication (e.g. configuration of the Octopus-Arm environment [15], [16]) is easy to develop thanks to provided tools ranging from simple XML stream processors (XmlReader and XmlWriter) to powerful LINQ to XML which allows comfortable operating on XML documents in an elegant and concise way.

- Network communication is also supported with easy to use highlevel classes and libraries like WCF (Windows Communication Foundation).

In any of these cases, user is required only to implement a wrapper class managing the interoperability using one of described mechanisms as a subclass of the Agent or Environment base class.

C. Comparison with RL-Glue

Although RL-Glue and dotRL are different types of solutions, some of their merits can be compared:

- RL-Glue being a network protocol is fully platform independent, whereas dotRL can work only on platforms with existing .NET framework implementation. However, the number of platforms which support .NET is becoming bigger. Thanks to the MONO project [31], .NET is supported not only on Windows, but also Unix-like systems, MacOS X, and even Android (full list of supported systems can be found in [32]).
- RL-Glue is however tied to the infrastructure: BigEndian convention is obligatory, and there are fixed sizes of basic data types. dotRL is completely infrastructure ignorant, as long as .NET Framework is supported.
- RL-Glue, being a handcrafted protocol is not feasible for extensibility (e.g. hard-coded order of elements in TaskSpec). dotRL is easily extensible by design.
- In both solutions adding new components requires at minimum writing only one class required to implement one simple interface.
- RL-Glue does not offer a standard way to turn off policy improvement. dotRL handles this case via “policy presentation” mode.
- In RL-Glue reporting and visualization must be managed by the user on their own (however, see RL-Viz project mentioned below). dotRL offers easily extensible reporting and presentation frameworks with convenient GUI.

RL-Glue is more popular, and in fact seems a better solution if multiplatform or distributed environment is obligatory. However, dotRL is easier to maintain (no hardcoded assumptions), provides more useful tools for evaluation of new agents (built in extensible reporting framework), visualization of environments (built in extensible visualization framework), and allows easy integration with other solutions (details were provided in sec. V-B). In the context of reporting and visualization it is fair to mention the RL-Viz project, however it remains unreleased since 2007 [33].

1BigEndian is obligatory in the context of network communication with the RL-Glue Core application. Refer to the RL-Glue Core implementation: “rlBufferWrite” and “rlBufferRead” functions in the “RL_network.c” file.
VI. CONCLUSIONS AND FUTURE WORK

In this paper dotRL — a platform for fast development and validation of reinforcement learning algorithms was introduced. The platform had been designed to minimize the time spent by its user on technical and infrastructural details, as they should focus on purely scientific issues. Seemingly, the platform meets this requirement.

Directions of further development of dotRL include its integration with other platforms. They also encompass enriching the set of agents and environments available within the platform.

REFERENCES


