Reinforcement Learning Library

**Date:** 2020-08-01

**Project:** ISO JTC1/SC22/WG21: Programming Language C++

**Audience:** SG19, WG21

**Authors:** Lawrence E. Lewis (SAS Institute)

Jorge Silva (SAS Institute)

Phil Ratzloff (SAS Institute)

Michael Wong (Codeplay)

**Contributors:** N/A

**Emails:** [Larry.Lewis@sas.com](mailto:Larry.Lewis@sas.com)

[Jorge.Silva@sas.com](mailto:Jorge.Silva@sas.com)

[Phil.Ratzloff@sas.com](mailto:Phil.Ratzloff@sas.com)

[michael@codeplay.com](mailto:michael@codeplay.com)

**Reply to:** [Larry.Lewis@sas.com](mailto:Larry.Lewis@sas.com)

Purpose and Scope

The purpose of this document is to present the most important data structures and algorithms required to create a useful reinforcement learning library. It does not attempt to be either mathematically rigorous or complete. It focuses on today’s most commonly used methods. While it presents some of the core concepts that underly reinforcement learning, it is not a tutorial.

Revision History

|  |  |
| --- | --- |
| **Revision** | **Description** |
| 1.0.0 | Initial version |

Impact on the Standard

A proposal derived from this document would be a ​**pure**​ library extension.

Concepts

The goal of reinforcement learning is to learn how to select a sequence of **actions** (a **policy**) that maximizes the **value** of the entire sequence. In order to learn this, it is assumed that right after each action is taken, the immediate value of taking the action (known as the **reward),** and resulting next **state** are known.

A common view of this process is that of an **agent** interacting with an **environment**. The agent would be the one using this library to do the learning and action selection. The environment may be real or simulated. At each **step**, the agent selects an action and sends that action to the environment. The environment applies that action, which may change the internal state of the environment, and the environment returns to the agent, both an immediate reward, and the new observable state**.** The entire internal state of the environment may not be available to the agent.

The agent is not focused on picking the action which maximizes the next short-term reward. Rather, the agent is interested in the long-term value, which is usually considered to be the **discounted sum** of the short-term rewards obtained by repeatedly selecting actions. The term **discounted** means that rewards received earlier in the process are considered more valuable than rewards received later. There is usually a discount factor that the agent uses to compute this.

While it may seem that the best way to maximize the sum of rewards is to maximize each individual reward, you have to remember that this is a state machine. Picking an action that maximizes the next short-term reward might put your environment in a state that prevents you from getting large rewards later on.

This library will be focused on data structures and algorithms needed by the software agent to learn how best to maximize the long-term value, rather than, for instance, the nuts and bolts of how to communicate with an environment.

The Action-Value Function: Q

In order to maximize the long term value, at each step, a trained agent selects the action that is most likely to maximize the long term value from this point forward. In order to do this, it needs to know the Action-Value function, (s,a), which returns the expected value (cumulative discounted reward) of doing action a in state s. This is the function that the learning algorithm must learn. Algorithms that learn the Q function are built using many of the common function estimation methods, but, currently, the most common is the **neural network**.

Data Structures for Reinforcement Learning

Since most current reinforcement learning algorithms depend on neural networks, the neural network is one of the key data structures required.

In turn, neural networks are usually represented as a **list** or **graph** of **layers**, where the layers contain, among other things, **multidimensional arrays** of numeric weights and gradients of the network output with respect to the weights**.**

Tensors

For machine learning, multidimensional arrays of numeric values are often referred to in the current literature as **tensors**, probably because of the use of this name in popular machine learning toolkits like Theano, TensorFlow, and PyTorch.

In physics, the term tensor has been used for over 120 years and is more than just a multidimensional array. It is a multidimensional array along with a specific set of operations (<https://en.wikipedia.org/wiki/Tensor>).

There are also some differences between a general multidimensional array, and a tensor as used in machine learning. Notably, machine learning tensors usually support a reshape operation, where, for instance, a 10 x 20 two dimensional image can be reshaped quickly to be a 200 x 1 vector so that the output of a convolutional or pooling layer in a neural network (which are often two dimensional images) can be fed into a fully-connected layer (which wants its input to be a one dimensional vector).

The C++ committee is currently considering the **mdspan** type to represent a multidimensional array ( <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0009r10.html> ). Also note that the linear algebra proposal ( <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1673r2.pdf> ) specifically excludes the definition of a Tensor (see page 11) for several reasons, while noting that “mdspan has natural use as a low-level representation of dense tensors”.

Algorithms for Reinforcement Learning

There are two categories of learning algorithms that need to exist in order to train a reinforcement learning agent.

First, since current reinforcement algorithms usually use neural networks as a core part of the overall algorithm, the standard neural network training algorithms must exist. These include standard **linear algebra** operations, like matrix multiply, the **backpropagation** algorithms that derive the gradient of the weights, and the **optimization** algorithms that find the weights that minimize the loss function (some function of the difference between the actual neural network output and the desired or target output). Hand-written back-propagation algorithms are often replaced by graph-based **automatic differentiation** algorithms.

Second, are the reinforcement learning specific algorithms. Currently, the most important of these are **DQN** and **Actor-Critic**. This is an active research area, however, and new algorithms are being frequently proposed.

One thing to note is how reinforcement learning differs from standard function optimization, such as that used in neural network training algorithms (or, for that matter, linear regression). The goal of standard function optimization is to find the parameters that minimize a parameterized function. For neural networks, the function to be minimized is usually the average difference, over a set of observations, of the output of the network and some known target value. The problem in reinforcement learning is that even when training, the observations DO NOT have a known target value. Even when reinforcement learning is using the **FQN** algorithm, which trains from a data set of recorded observations, the selected action and resulting next state that are recorded in that data set are NOT necessarily the optimal action that could have been selected. It is just the action that was selected by the agent being observed at that time.

The Environment

While a reinforcement learning environment could be anything that responds in the proper way to an agent (receives an action and returns an immediate reward and next state), in practice the software interface to a reinforcement learning environment is actually becoming pretty standardized.

Many reinforcement learning packages implement much of the interface that was defined by the Python based openai Gym toolkit: <https://gym.openai.com/docs/>.

I recommend that something very similar should be adopted by C++, except, of course, that the objects be strongly typed. In my experience so far, there is no need to discover any of the types.

The openai Gym toolkit defines the following objects (in Python):

**Action Space**: The actions available to an agent. These can be discrete (left, right, up, down) or continuous (set velocity to 10.2m/sec).

**Observation Space**: The description of the observable state available to an agent, such as the number and type of each of the state variables.  
  
**Observation**: The current observable state of the environment.

**Reward**: A floating point value that is the reward obtained from applying the last selected action.

**Done**: A Boolean flag indicating the environment has reached a terminal state.  
  
And, openai Gym defines the following methods of the environment:

**Reset**: initial\_observation = environment.reset()

**Step**: observation, reward, done, debug\_info = environment.step(action)

What should be standardized now

The first thing that needs to be standardized are the algorithms and data structures and that reinforcement learning depends on. To summarize:

* Multidimensional Arrays (including machine learning Tensor operations)
* Graphs (needed by Automatic Differentiation and Neural Nets)
* Linear Algebra
* Optimization (algorithms like Stochastic Gradient Descent, LBFGS, etc.)
* Automatic Differentiation
* Neural Networks