Reinforcement Learning in R

Nicolas Pröllochs

2020-03-02

This vignette gives an introduction to the `ReinforcementLearning` package, which allows one to perform model-free reinforcement in *R*. The implementation uses input data in the form of sample sequences consisting of states, actions and rewards. Based on such training examples, the package allows a reinforcement learning agent to learn an optimal policy that defines the best possible action in each state. In the following sections, we present multiple step-by-step examples to illustrate how to take advantage of the capabilities of the `ReinforcementLearning` package. Moreover, we present methods to customize the learning and action selection behavior of the agent. Main features of `ReinforcementLearning` include, but are not limited to:

- Learning an optimal policy from a fixed set of a priori known transition samples
- Predefined learning rules and action selection modes
- A highly customizable framework for model-free reinforcement learning tasks

Introduction

Reinforcement learning refers to the problem of an agent that aims to learn optimal behavior through trial-and-error interactions with a dynamic environment. All algorithms for reinforcement learning share the property that the feedback of the agent is restricted to a reward signal that indicates how well the agent is behaving. In contrast to supervised machine learning methods, any instruction concerning how to improve its behavior is absent. Thus, the goal and challenge of reinforcement learning is to improve the behavior of an agent given only this limited type of feedback.

The reinforcement learning problem

In reinforcement learning, the decision-maker, i.e. the agent, interacts with an environment over a sequence of observations and seeks a reward to be maximized over time. Formally, the model consists of a finite set of environment states SS , a finite set of agent actions AA , and a set of scalar reinforcement signals (i.e. rewards) RR . At each iteration ii , the agent observes some representation of the environment's state $s_i \in S$si∈S . On that basis, the agent selects an action $a_i \in A(s_i)$ai∈A(si) , where $A(s_i) \subseteq A$A(si)⊆A denotes the set of actions available in state $s_i$si . After each iteration, the agent receives a numerical reward $r_{i+1} \in R$ri+1∈R and observes a new state s_{i+1}si+1 .

Policy learning

In order to store current knowledge, the reinforcement learning method introduces a so-called state-action function $Q(s_i,a_i)$Q(si,ai) , which defines the expected value of each possible action $a_i$ai in each state $s_i$si . If $Q(s_i,a_i)$Q(si,ai) is known, then the optimal policy $\pi_*(s_i,a_i)$π∗(si,ai) is given by the action $a_i$ai , which maximizes $Q(s_i,a_i)$Q(si,ai) given the state $s_i$si . Consequently, the learning problem of the agent is to maximize the expected reward by learning an optimal policy function $\pi_*(s_i,a_i)$π∗(si,ai) .

Experience replay

Experience replay allows reinforcement learning agents to remember and reuse experiences from the past. The underlying idea is to speed up convergence by replaying observed state transitions repeatedly to the agent, as if they were new observations collected while interacting with a system. Hence, experience replay only requires input data in the form of sample sequences consisting of states, actions and rewards. These data points can be, for example, collected from a running system without the need for direct interaction. The stored training examples then allow the agent to learn a state-action function and an optimal policy for every state transition in the input data. In a next step, the policy can be applied to the system for validation purposes or to collect new data points (e.g. in order to iteratively improve the current policy). As its main advantage, experience replay can speed up convergence by allowing for the back-propagation of information from updated states to preceding states without further interaction.

Setup of the ReinforcementLearning package

Even though reinforcement learning has recently gained a great deal of traction in studies that perform human-like learning, the available tools are not living up to the needs of researchers and practitioners. The `ReinforcementLearning` package is intended to partially close this gap and offers the ability to perform model-free reinforcement learning in a highly customizable framework.

Installation

Using the `devtools` package, one can easily install the latest development version of `ReinforcementLearning` as follows.

```
# install.packages("devtools")

# Option 1: download and install latest version from GitHub
devtools::install_github("nproellochs/ReinforcementLearning")

# Option 2: install directly from bundled archive
devtoos::install_local("ReinforcementLearning_1.0.0.tar.gz")
```

Package loading

Afterwards, one merely needs to load the `ReinforcementLearning` package as follows.

```
library(ReinforcementLearning)
```

Usage

The following sections present the usage and main functionality of the `ReinforcementLearning` package.

Data preparation

The `ReinforcementLearning` package utilizes different mechanisms for reinforcement learning, including Q-learning and experience replay. It thereby learns an optimal policy based on past experience in the form of sample sequences consisting of states, actions and rewards. Consequently, each training example consists of a state-transition tuple $(s_i, a_i, r_{i+1}, s_{i+1})$(si,ai,ri+1,si+1) as follows:

- $s_i$si is the current environment state.
- $a_i$ai denotes the selected action in the current state.
- r_{i+1}ri+1 specifies the immediate reward received after transitioning from the current state to the next state.
- s_{i+1}si+1 refers to the next environment state.

The training examples for reinforcement learning can (1) be collected from an external source and inserted into a tabular data structure, or (2) generated dynamically by querying a function that defines the behavior of the environment. In both cases, the corresponding input must follow the same tuple structure $(s_i, a_i, r_{i+1}, s_{i+1})$(si,ai,ri+1,si+1) . We detail both variants in the following.

Learning from pre-defined observations

This approach is beneficial when the input data is pre-determined or one wants to train an agent that replicates past behavior. In this case, one merely needs to insert a tabular data structure with past observations into the package. This might be the case when the state-transition tuples have been collected from an external source, such as sensor data, and one wants to learn an agent by eliminating further interaction with the environment.

The following example shows the first five observations of a representative dataset containing game states of randomly sampled tic-tac-toe games. In this dataset, the first column contains a representation of the current board state in a match. The second column denotes the observed action of player X in this state, whereas the third column contains a representation of the resulting board state after performing the action. The fourth column specifies the resulting reward for player X. This dataset is thus sufficient as input for learning the agent.

```
data("tictactoe")
head(tictactoe, 5)

##       State Action NextState Reward

## 1 .........     c7 ......X.B      0

## 2 ......X.B     c6 ...B.XX.B      0

## 3 ...B.XX.B     c2 .XBB.XX.B      0

## 4 .XBB.XX.B     c8 .XBBBXXXB      0

## 5 .XBBBXXXB     c1 XXBBBXXXB      0
```

Dynamic learning from an interactive environment function

An alternative strategy is to define a function that mimics the behavior of the environment. One can then learn an agent that samples experience from this function. Here the environment function takes a state-action pair as input. It then returns a list containing the name of the next state and the reward. In this case, one can also utilize R to access external data sources, such as sensors, and execute actions via common interfaces. The structure of such a function is represented by the following pseudocode:

```
environment <- function(state, action) {

  ...

  return(list("NextState" = newState,
```

```
              "Reward" = reward))

}
```

After specifying the environment function, we can use to collect random sequences from it. Thereby, the input specifies number of samples (NN), the environment function, the set of states (i.e. SS) and the set of actions (i.e. AA). The return value is then a data frame containing the experienced state transition tuples $(s_i,a_i,r_{i+1},s_{i+1})$(si,ai,ri+1,si+1) for $i=1,...,N$i=1,…,N . The following code snippet shows how to generate experience from an exemplary environment function.

```
# Define state and action sets
states <- c("s1", "s2", "s3", "s4")
actions <- c("up", "down", "left", "right")

env <- gridworldEnvironment

# Sample N = 1000 random sequences from the environment
data <- sampleExperience(N = 1000,
                         env = env,
                         states = states,
                         actions = actions)
```

Learning phase

General setup

The routine `ReinforcementLearning()` bundles the main functionality, which teaches a reinforcement learning agent using the previous input data. For this purpose, it requires the following arguments: (1) A *data* argument that must be a data frame object in which each row represents a state transition tuple $(s_i,a_i,r_{i+1},s_{i+1})$(si,ai,ri+1,si+1) . (2) The user is required to specify the column names of the individual tuple elements within *data*.

The following pseudocode demonstrates the usage for pre-defined data from an external source, while the subsequent sections detail the interactive setup. Here the parameters *s*, *a*, *r* and *s_new* contain strings specifying the corresponding column names in the data frame *data*.

```
# Load dataset
data("tictactoe")

# Perform reinforcement learning
model <- ReinforcementLearning(data = tictactoe,
                               s = "State",
                               a = "Action",
                               r = "Reward",
                               s_new = "NextState",
                               iter = 1)
```

Parameter configuration

Several parameters can be provided to in order to customize the learning behavior of the agent.

- **alpha** The learning rate, set between 0 and 1. Setting it to 0 means that the Q-values are never updated and, hence, nothing is learned. Setting a high value, such as 0.9, means that learning can occur quickly.
- **gamma** Discount factor, set between 0 and 1. Determines the importance of future rewards. A factor of 0 will render the agent short-sighted by only considering current rewards, while a factor approaching 1 will cause it to strive for a greater reward over the long run.

- **epsilon** Exploration parameter, set between 0 and 1. Defines the exploration mechanism in ε -greedy action selection. In this strategy, the agent explores the environment by selecting an action at random with probability ε . Alternatively, the agent exploits its current knowledge by choosing the optimal action with probability $1-\varepsilon$. This parameter is only required for sampling new experience based on an existing policy.
- **iter** Number of repeated learning iterations the agent passes through the training dataset. Iter is an integer greater than 0. The default is set to 1 in which each state transition tuple is presented to the agent only once. Depending on the size of the training data, a higher number of repeated learning iterations can improve convergence but requires longer computation time. This parameter is passed directly to `ReinforcementLearning()`.

The learning parameters **alpha**, **gamma**, and **epsilon** must be provided in an optional **control** object passed to the `ReinforcementLearning()` function.

```
# Define control object
control <- list(alpha = 0.1, gamma = 0.1, epsilon = 0.1)

# Pass learning parameters to reinforcement learning function
model <- ReinforcementLearning(data, iter = 10, control = control)
```

Diagnostics

The result of the learning process is an object of type `rl` that contains the state-action table and an optimal policy with the best possible action in each state. The command `computePolicy(model)` shows the optimal policy, while `print(model)` outputs the state-action table, i.e. the Q-value of each state-action pair. In addition, `summary(model)` prints further model details and summary statistics.

```
# Print policy
computePolicy(model)

# Print state-action table
print(model)

# Print summary statistics
summary(model)
```

Working example 1: Gridworld

This section demonstrates the capabilities of the `ReinforcementLearning` package with the help of a practical example.

Problem definition

Our practical example aims at teaching optimal movements to a robot in a grid-shaped maze (adapted from Sutton (1998)). Here the agent must navigate from a random starting position to a final position on a simulated 2×2 grid (see figure below). Each cell on the grid reflects one state, yielding a total of 4 different states. In each state, the agent can perform one out of four possible actions, i.e. to move up, down, left, or right, with the only restriction being that it must remain on the grid. In other words, the grid is surrounded by a wall, which makes it impossible for the agent to move off the grid. A wall between *s1* and *s4* hinders direct movements between these states. Finally, the reward structures is as follows: each movement leads to a negative reward of -1 in order to penalize routes that are not the shortest path. If the agent reaches the goal position, it earns a reward of 10.

|————|
| s1 | s4 |
| s2 s3 |
|————|

Defining an environment function

We first define the sets of available states (*states*) and actions (*actions*).

```
# Define state and action sets
states <- c("s1", "s2", "s3", "s4")
actions <- c("up", "down", "left", "right")
```

We then rewrite the above problem formulation into the following environment function. As previously mentioned, this function must take a state and an action as input. The if-conditions determine the current combination of state and action. In our example, the state refers to the agent's position on the grid and the action denotes the intended movement. Based on these, the function decides upon the next state and a numeric reward. These together are returned as a list.

```
# Load built-in environment function for 2x2 gridworld
env <- gridworldEnvironment
print(env)
```

```
## function (state, action)
## {
##     next_state <- state
##     if (state == state("s1") && action == "down")
##         next_state <- state("s2")
##     if (state == state("s2") && action == "up")
##         next_state <- state("s1")
##     if (state == state("s2") && action == "right")
##         next_state <- state("s3")
##     if (state == state("s3") && action == "left")
##         next_state <- state("s2")
##     if (state == state("s3") && action == "up")
##         next_state <- state("s4")
##     if (next_state == state("s4") && state != state("s4")) {
##         reward <- 10
##     }
##     else {
##         reward <- -1
##     }
##     out <- list(NextState = next_state, Reward = reward)
##     return(out)
## }
## <bytecode: 0x000000001823a148>
## <environment: namespace:ReinforcementLearning>
```

Learning an optimal policy

After having specified the environment function, we can use the built-in `sampleExperience()` function to sample observation sequences from the environment. The following code snippet generates a data frame *data* containing 1000 random state-transition tuples $(s_i, a_i, r_{i+1}, s_{i+1})$(si,ai,ri+1,si+1) .

```
# Sample N = 1000 random sequences from the environment
data <- sampleExperience(N = 1000,
                         env = env,
                         states = states,
                         actions = actions)
head(data)
```

```
##   State Action Reward NextState
## 1    s4   left     -1        s4
## 2    s2  right     -1        s3
## 3    s2  right     -1        s3
## 4    s3   left     -1        s2
## 5    s4     up     -1        s4
## 6    s1   down     -1        s2
```

We can now use the observation sequence in *data* in order to learn the optimal behavior of the agent. For this purpose, we first customize the learning behavior of the agent by defining a control object. We follow the default parameter choices and set the learning rate *alpha* to 0.1, the discount factor *gamma* to 0.5 and the exploration greediness *epsilon* to 0.1. Subsequently, we use the `ReinforcementLearning()` function to learn the best possible policy for the the input data.

```
# Define reinforcement learning parameters
control <- list(alpha = 0.1, gamma = 0.5, epsilon = 0.1)

# Perform reinforcement learning
model <- ReinforcementLearning(data,
                               s = "State",
                               a = "Action",
                               r = "Reward",
                               s_new = "NextState",
                               control = control)
```

Evaluating policy learning

The `ReinforcementLearning()` function returns an `rl` object. We can evoke `computePolicy(model)` in order to display the policy that defines the best possible action in each state. Alternatively, we can use `print(model)` in order to write the entire state-action table to the screen, i.e. the Q-value of each state-action pair. Evidently, the agent has learned the optimal policy that allows it to take the shortest path from an arbitrary starting position to the goal position *s4*.

```
# Print policy
computePolicy(model)
```

```
##      s1      s2      s3      s4
##  "down" "right"    "up" "right"
```

```
# Print state-action function
print(model)
```

```
## State-Action function Q
##         right         up       down       left
## s1 -0.6633782 -0.6687457  0.7512191 -0.6572813
```

```
## s2  3.5806843 -0.6893860  0.7760491  0.7394739
## s3  3.5702779  9.1459425  3.5765323  0.6844573
## s4 -1.8005634 -1.8567931 -1.8244368 -1.8377018
##
## Policy
##      s1      s2      s3      s4
##  "down" "right"    "up" "right"
##
## Reward (last iteration)
## [1] -263
```

Ultimately, we can use `summary(model)` to inspect the model further. This command outputs additional diagnostics regarding the model such as the number of states and actions. Moreover, it allows us to analyze the distribution of rewards. For instance, we see that the total reward in our sample (i.e. the sum of the rewards column *r*r) is highly negative. This indicates that the random policy used to generate the state transition samples deviates from the optimal case. Hence, the next section explains how to apply and update a learned policy with new data samples.

```
# Print summary statistics
summary(model)
```

```
##
## Model details
## Learning rule:           experienceReplay
## Learning iterations:     1
## Number of states:        4
## Number of actions:       4
## Total Reward:            -263
##
## Reward details (per iteration)
## Min:                     -263
## Max:                     -263
## Average:                 -263
## Median:                  -263
## Standard deviation:      NA
```

Applying a policy to unseen data

We now apply an existing policy to unseen data in order to evaluate the out-of-sample performance of the agent. The following example demonstrates how to sample new data points from an existing policy. The result yields a column with the best possible action for each given state.

```
# Example data
data_unseen <- data.frame(State = c("s1", "s2", "s1"),
                          stringsAsFactors = FALSE)

# Pick optimal action
data_unseen$OptimalAction <- predict(model, data_unseen$State)
```

```
data_unseen
```

```
##   State OptimalAction
## 1    s1          down
## 2    s2         right
## 3    s1          down
```

Updating an existing policy

Finally, one can update an existing policy with new observational data. This is beneficial when, for instance, additional data points become available or when one wants to plot the reward as a function of the number of training samples. For this purpose, the `ReinforcementLearning()` function can take an existing `rl` model as an additional input parameter. Moreover, it comes with an additional pre-defined action selection mode, namely $\varepsilon\varepsilon$ -greedy, thereby following the best action with probability $1-\varepsilon 1-\varepsilon$ and a random one with $\varepsilon\varepsilon$.

```
# Sample N = 1000 sequences from the environment
# using epsilon-greedy action selection
data_new <- sampleExperience(N = 1000,
                             env = env,
                             states = states,
                             actions = actions,
                             actionSelection = "epsilon-greedy",
                             model = model,
                             control = control)

# Update the existing policy using new training data
model_new <- ReinforcementLearning(data_new,
                                   s = "State",
                                   a = "Action",
                                   r = "Reward",
                                   s_new = "NextState",
                                   control = control,
                                   model = model)
```

The following code snippet shows that the updated policy yields significantly higher rewards as compared to the previous policy. These changes can also be visualized in a learning curve via `plot(model_new)`.

```
# Print result
print(model_new)
```

```
## State-Action function Q
```

```
##        right         up       down       left
## s1 -0.643587 -0.6320560  0.7657318 -0.6314927
## s2  3.530829 -0.6407675  0.7714129  0.7427914
## s3  3.548196  9.0608344  3.5521760  0.7382102
## s4 -1.939574 -1.8922783 -1.8835278 -1.8856132
```

```
##
```

```
## Policy
```

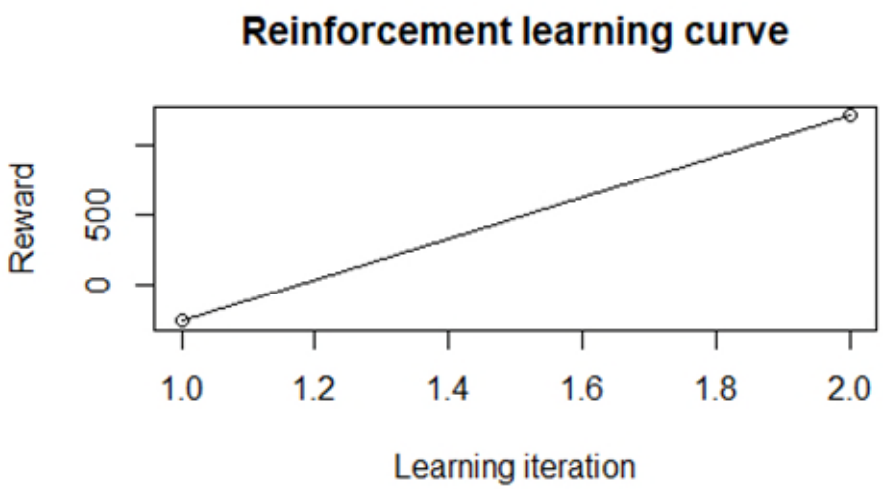
```
##      s1      s2      s3      s4
##  "down" "right"    "up"  "down"
```

```
##
```

```
## Reward (last iteration)
```

```
## [1] 1211
```

```r
# Plot reinforcement learning curve
plot(model_new)
```



Working example 2: Tic-Tac-Toe

This section demonstrates the capabilities of the `ReinforcementLearning` package when using state-transition tuples from an external source without the need for modeling the dynamics of the environment.

Problem definition

The following example utilizes the aforementioned dataset containing 406,541 game states of Tic-Tac-Toe to learn the optimal actions for each state of the board (adapted from Sutton (1998)). All states are observed from the perspective of player X who is also assumed to have played first. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game. Reward for player X is +1 for 'win', 0 for 'draw', and -1 for 'loss'..

The current state of the board is represented by a rowwise concatenation of the players' marks in a 3x3 grid. For example, "……X.B" denotes a board state in which player X has placed a mark in the first field of the third column whereas player B has placed a mark in the third field of the third column (see visualization below).

```
## ......X.B
## |  .  |  .  |  .  |
## |-----------------|
## |  .  |  .  |  .  |
## |-----------------|
## |  X  |  .  |  B  |
```

The following code utilizes the dataset to learn the optimal actions for each state of the board. The computation time is 1-2 minutes on standard desktop computers.

```r
# Load dataset
data("tictactoe")

# Define reinforcement learning parameters
control <- list(alpha = 0.2, gamma = 0.4, epsilon = 0.1)
```

```r
# Perform reinforcement learning
model <- ReinforcementLearning(tictactoe, s = "State", a = "Action", r = "Reward",
                               s_new = "NextState", iter = 1, control = control)

# Calculate optimal policy
pol <- computePolicy(model)

# Print policy
head(pol)
```

```
## .XXBB..XB XXBB.B.X. .XBB..BXX BXX...B.. ..XB..... XBXBXB...

##      "c1"      "c5"      "c5"      "c4"      "c5"      "c9"
```

The `ReinforcementLearning()` function returns an `rl` object. Subsequently, evoking `computePolicy(model)` returns a named vector which allows one to display the policy that defines the best possible action in each state. Here, the names represent the state names (representation of the current board in a match) while the values denote the optimal action. For example, in state ".XXBB..XB" (see grid below), the optimal action for the agent is to place a mark in "c1".

```
## |  .  |  X  |  X   |

## |-----------------|

## |  B  |  B  |  .   |

## |-----------------|

## |  .  |  X  |   B  |

## |  c1  |  c2  |  c3   |

## |--------------------|

## |  c4  |  c5  |  c6   |

## |--------------------|

## |  c7  |  c8  |   c9  |
```

Notes on performance

Q-learning is guaranteed to converge to an optimal policy. However, the method is computationally demanding as it relies on continuous interactions between an agent and its environment. To remedy this, the `ReinforcementLearning` package allows users to perform batch reinforcement learning. In most scenarios, this reinforcement learning variant benefits computational performance as it mitigates the 'exploration overhead' problem in pure online learning. In combination with experience replay, it speeds up convergence by collecting and replaying observed state transitions repeatedly to the agent as if they were new observations collected while interacting with the system. Nonetheless, due to the fact that the package is written purely in R, the applicability of the package to very large scale problems (such as applications from computer vision) is still limited. In the following, we briefly summarize scenarios the package is capable of handling and situations in which one should consider utilizing reinforcement learning implementations written in "faster" programming languages.

What the `ReinforcementLearning` R package can do:

- Learning optimal strategies for real-world problems with limited state and action sets (e.g. finding optimal strategies for simple games, training a simple stock market trading agent, learning polarity labels in applications from natural language processing).
- The packages allows one to speed up performance by adjusting learning parameters and making use of experience replay.
- The package allows one to train an agent from pre-defined observations without the need for modeling the dynamics of the environment. Typically, this approach drastically speeds

up convergence and can be useful in situations in which the state-transition tuples have been collected from an external source, such as sensor data.
- The package provides a highly customizable framework for model-free reinforcement learning tasks in which the functionality can easily be extended. For example, users may attempt to speed up performance by defining alternative reinforcement learning algorithms and integrating them into the package code.

What the `ReinforcementLearning` R package cannot do:

- Solving large-scale problems with high-dimensional state-action spaces such as those from computer vision (users may consider reinforcement learning implementations written in “faster” programming languages)
- Solving reinforcement learning problems requiring real-time interaction (e.g. real-time interaction with a robot)