ReinforcementLearning: A package for replicating human behavior in R

NICOLAS PROELLOCHS AND STEFAN FEUERRIEGEL

2017-04-06

Introduction

Reinforcement learning has recently gained a great deal of traction in studies that call for human-like learning. In settings where an explicit teacher is not available, this method teaches an agent via interaction with its environment without any supervision other than its own decision-making policy. In many cases, this approach appears quite natural by mimicking the fundamental way humans learn. However, implementing reinforcement learning is programmatically challenging, since it relies on continuous interactions between an agent and its environment. In fact, there is currently no package available that performs model-free reinforcement learning in *R*. As a remedy, we introduce the `ReinforcementLearning` *R* package, which allows an agent to learn optimal behavior based on sample experience consisting of states, actions and rewards. The result of the learning process is a highly interpretable reinforcement learning 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

Reinforcement learning

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 S, a finite set of agent actions A, and a set of scalar reinforcement signals (i.e. rewards) R. At each iteration i, the agent observes some representation of the environment's state $s_i \in S$. On that basis, the agent selects an action $a_i \in A(s_i)$, where $A(s_i) \subseteq A$ denotes the set of actions available in state s_i. After each iteration, the agent receives a numerical reward $r_{i+1} \in R$ and observes a new state s_{i+1}.

Policy learning

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

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 format

The `ReinforcementLearning` package uses experience replay to learn an optimal policy based on past experience in the form of sample sequences consisting of states, actions and rewards. Here each training example consists of a state transition tuple *(s,a,r,s_new)*, as described ibelow.

- **s** The current environment state.
- **a** The selected action in the current state.
- **r** The immediate reward received after transitioning from the current state to the next state.
- **s_new** The next environment state.

Note:

- The input data must be a dataframe in which each row represents a state transition tuple *(s,a,r,s_new)*.

Read-in sample experience

The state transition tuples can be collected from an external source and easily read-in into *R*. The sample experience can then be used to train a reinforcement learning agent without requiring further interaction with the environment. The following example shows a representative dataset containing game states of 100,000 randomly sampled Tic-Tac-Toe games.

```
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
```

Experience sampling using an environment function

The `ReinforcementLearning` package is shipped with the built-in capability to sample experience from a function that defines the dynamics of the environment. If the dynamics of the environment are known a priori, one can set up an arbitrary complex environment function in *R* and sample state transition tuples. This function has to be manually implemented and must take a state and an action as input. The return value must be a list containing the name of the next state and the reward. As a main advantage, this method of experience sampling allows one to easily validate the performance of reinforcement learning, by applying the learned policy to newly generated samples.

```
 environment <- function(state, action) {
   ...
   return(list("NextState" = newState,
               "Reward" = reward))
}
```

The following example illustrates how to generate sample experience using an environment function. Here we collect experience from an agent that navigates from a random starting position to a goal position on a simulated 2×2 grid (see figure below).

```
|———-|
| s1 | s4 |
| s2   s3 |
|———-|
```

Each cell on the grid represents one state, which yields a total of 4 states. The grid is surrounded by a wall, which makes it impossible for the agent to move off the grid. In addition, the agent faces a wall between s1 and s4. At each state, the agent randomly chooses one out of four possible actions, i. e. to move up, down, left, or right. The agent encounters the following reward structure: Crossing each square on the grid leads to a reward of -1. If the agent reaches the goal position, it earns a reward of 10.

```
 # Load exemplary environment (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)
## }
## <environment: namespace:ReinforcementLearning>
```

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

# 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
```

Performing reinforcement learning

The following example shows how to teach a reinforcement learning agent using input data in the form of sample sequences consisting of states, actions and rewards. The ‘data’ argument must be a dataframe object in which each row represents a state transition tuple *(s,a,r,s_new)*. Moreover, the user is required to specify the column names of the individual tuple elements in ‘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)

# Print result
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
```

The result of the learning process is a state-action table and an optimal policy that defines the best possible action in each state.

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

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

Updating an existing policy

Specifying an environment function to model the dynamics of the environment allows one to easily validate the performance of the agent. In order to do this, one simply applies the learned policy to newly generated samples. For this purpose, the `ReinforcementLearning` package comes with an additional predefined action selection mode, namely 'epsilon-greedy'. 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 $1-\varepsilon$.

The following example shows how to sample new experience from an existing policy using 'epsilon-greedy' action selection. The result is new state transition samples ('data_new') with significantly higher rewards compared to the original sample ('data').

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

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

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

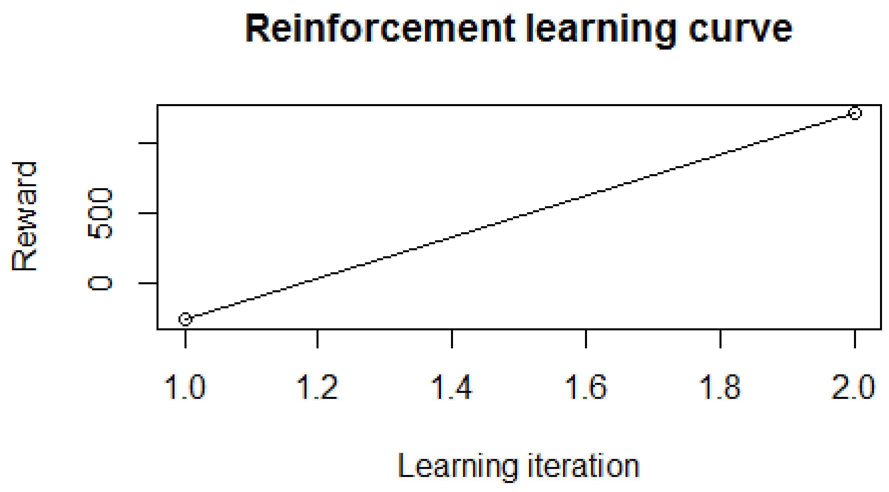
```
# Update the existing policy using new training data
model_new <- ReinforcementLearning(data_new, s = "State", a = "Action", r =
"Reward", s_new = "NextState", control = control, model = model)

# 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
```

```
summary(model_new)
```

```
##
## Model details
## Learning rule:           experienceReplay
## Learning iterations:     2
## Number of states:        4
## Number of actions:       4
## Total Reward:            1211
##
## Reward details (per iteration)
## Min:                     -263
## Max:                     1211
## Average:                 474
## Median:                  474
## Standard deviation:      1042.275
```

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



Parameter configuration

The `ReinforcementLearning` package allows for the adjustment of the following parameters 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 term.
- **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. Iter is an integer greater than 0. The default is set to 1.

```
 # 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)
```

Working example: Learning Tic-Tac-Toe

The following example shows the use of `ReinforcementLearning` in an applied setting. More precisely, we utilize a dataset containing 406,541 game states of Tic-Tac-Toe to learn the optimal actions for each state of the board.

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

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

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

# Print optimal policy
policy(model)
```

Notes:

- 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'.

Download: https://github.com/nproellochs/ReinforcementLearning

blogisr / 8. April 2017