Quality and Innovation

PERFORMANCE EXCELLENCE & DATA SCIENCE FOR DIGITAL TRANSFORMATION IN INDUSTRY 4.0

SATURDAY, MARCH 24TH, 2018

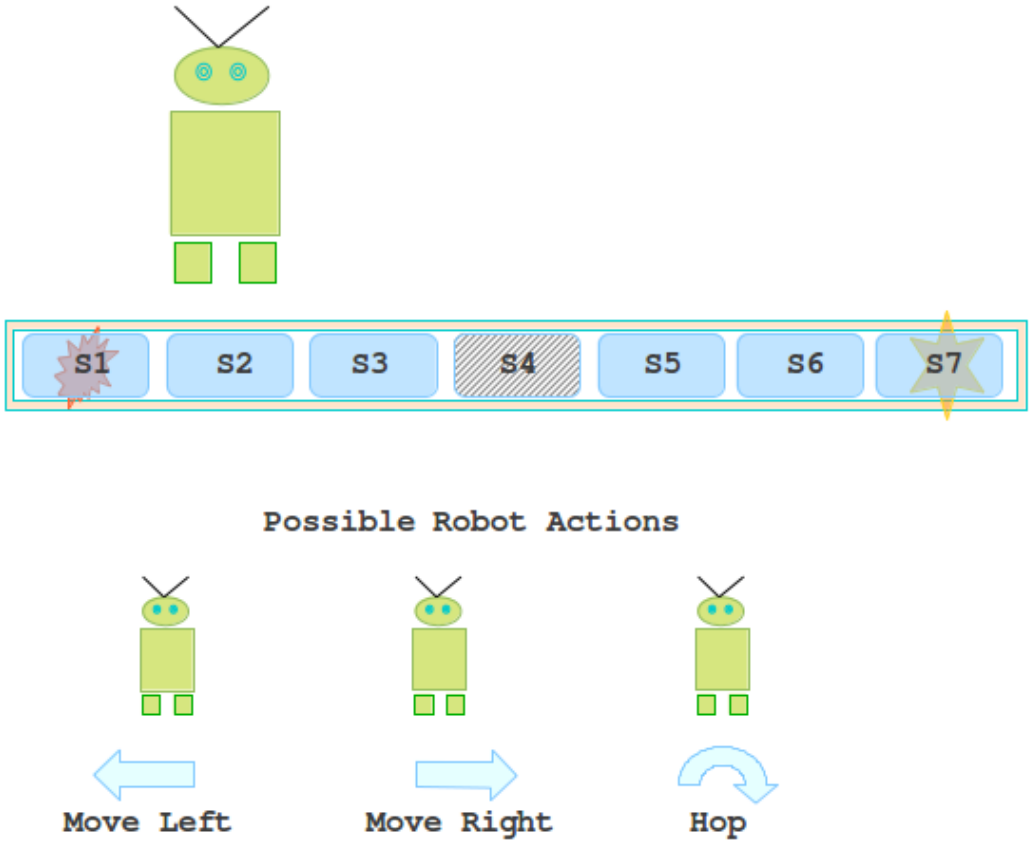
DATA SCIENCE

Reinforcement Learning: Q-Learning with the Hopping Robot

BY NICOLE RADZIWILL ON DECEMBER 24, 2017 • (5 COMMENTS)

Overview: Reinforcement learning uses "reward" signals to determine how to navigate through a system in the most valuable way. (I'm particularly interested in the variant of reinforcement learning called "Q-Learning" because the goal is to create a "Quality Matrix" that can help you make the best sequence of decisions!) I found a toy robot navigation problem on the web that was solved using custom R code for reinforcement learning, and I wanted to reproduce the solution in different ways than the original author did. This post describes different ways that I solved the problem described at http://bayesianthink.blogspot.com/2014/05/hopping-robots-and-reinforcement.html (http://bayesianthink.blogspot.com/2014/05/hopping-robots-and-reinforcement.html)

The Problem: Our *agent*, the robot, is placed at random on a board of wood. There's a hole at s1, a sticky patch at s4, and the robot is trying to make appropriate decisions to navigate to s7 (the target). The image comes from the blog post linked above.



To solve a problem like this, you can use MODEL-BASED approaches if you know how likely it is that the robot will move from one state to another (that is, the *transition probabilities* for each action) or MODEL-FREE approaches (you don't know how likely it is that the robot will move from state to state, but you *can* figure out a reward structure).

- **Markov Decision Process (MDP)** – If you know the states, actions, rewards, and transition probabilities (which are probably different for each action), you can determine the optimal policy or "path" through the system, given different starting states. (If transition probabilities have nothing to do with decisions that an agent makes, your MDP reduces to a **Markov Chain**.)
- **Reinforcement Learning (RL)** – If you know the states, actions, and rewards (but not the transition probabilities), you can still take an unsupervised approach. Just randomly create lots of hops through your system, and use them to update a matrix that describes the average value of each hop within the context of the system.

Solving a RL problem involves finding the optimal value functions (e.g. the Q matrix in Attempt 1) or the optimal policy (the State-Action matrix in Attempt 2). Although there are many techniques for reinforcement learning, we will use Q-learning because we don't know the transition probabilities for each action. (If we did, we'd model it as a Markov Decision Process and use the MDPtoolbox package instead.) Q-Learning relies on traversing the system in many ways to update a matrix of average expected rewards from each state transition. This equation that it uses is from https://www.is.uni-freiburg.de/ressourcen/business-

analytics/13_reinforcementlearning.pdf (https://www.is.uni-freiburg.de/ressourcen/business-analytics/13_reinforcementlearning.pdf):

$$Q(s,a) \leftarrow \underbrace{Q(s,a)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \left[\underbrace{r'}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_{a'} Q(s',a')}_{\text{expected optimal value}} - \underbrace{Q(s,a)}_{\text{old value}} \right]$$

For this to work, *all states* have to be visited a sufficient number of times, and *all state-action pairs* have to be included in your experience sample. So keep this in mind when you're trying to figure out how many iterations you need.

Attempt 1: Quick Q-Learning with qlearn.R

- **Input**: A rewards matrix R. (That's all you need! Your states are encoded in the matrix.)
- **Output**: A Q matrix from which you can extract optimal policies (or paths) to help you navigate the environment.
- **Pros**: Quick and *very easy.* **Cons**: Does not let you set epsilon (% of random actions), so all episodes are determined randomly and it may take longer to find a solution. Can take a long time to converge.

Set up the rewards matrix so it is a square matrix with *all the states down the rows, starting with the first* and *all the states along the columns, starting with the first:*

```
hopper.rewards <- c(-10, 0.01, 0.01, -1, -1, -1, -1,
         -10, -1, 0.1, -3, -1, -1, -1,
         -1, 0.01, -1, -3, 0.01, -1, -1,
         -1, -1, 0.01, -1, 0.01, 0.01, -1,
         -1, -1, -1, -3, -1, 0.01, 100,
         -1, -1, -1, -1, 0.01, -1, 100,
         -1, -1, -1, -1, -1, 0.01, 100)

HOP <- matrix(hopper.rewards, nrow=7, ncol=7, byrow=TRUE)
> HOP
     [,1]  [,2]  [,3] [,4]  [,5]  [,6] [,7]
[1,]  -10  0.01  0.01   -1 -1.00 -1.00   -1
[2,]  -10 -1.00  0.10   -3 -1.00 -1.00   -1
[3,]   -1  0.01 -1.00   -3  0.01 -1.00   -1
[4,]   -1 -1.00  0.01   -1  0.01  0.01   -1
[5,]   -1 -1.00 -1.00   -3 -1.00  0.01  100
[6,]   -1 -1.00 -1.00   -1  0.01 -1.00  100
[7,]   -1 -1.00 -1.00   -1 -1.00  0.01  100
```

Here's how you read this: the rows represent where you've come FROM, and the columns represent where you're going TO. Each element 1 through 7 corresponds directly to S1 through S7 in the cartoon above. Each cell contains a reward (or penalty, if the value is negative) if we arrive in that state.

The S1 state is *bad* for the robot… there's a hole in that piece of wood, so we'd really like to keep it away from that state. Location [1,1] on the matrix tells us what reward (or penalty) we'll receive if we start at S1 and *stay* at S1: -10 (that's bad). Similarly, location [2,1] on the matrix tells us that if we start at S2 and move left to S1, that's also bad and we should receive a penalty of -10. The S4 state is also undesirable –

there's a sticky patch there, so we'd like to keep the robot away from it. Location [3,4] on the matrix represents the action of going from S3 to S4 by moving right, which will put us on the sticky patch

Now load the qlearn command into your R session:

```
qlearn <- function(R, N, alpha, gamma, tgt.state) {
# Adapted from https://stackoverflow.com/questions/39353580/how-
  Q <- matrix(rep(0,length(R)), nrow=nrow(R))
  for (i in 1:N) {
    cs <- sample(1:nrow(R), 1)
    while (1) {
      next.states <- which(R[cs,] > -1)  # Get feasible actions
      if (length(next.states)==1)        # There may only be one
        ns <- next.states
      else
        ns <- sample(next.states,1) # Or you may have to pick fr
      if (ns > nrow(R)) { ns <- cs }
      # NOW UPDATE THE Q-MATRIX
      Q[cs,ns] <- Q[cs,ns] + alpha*(R[cs,ns] + gamma*max(Q[ns, w
      if (ns == tgt.state) break
      cs <- ns
    }
  }
  return(round(100*Q/max(Q)))
}
```

Run qlearn with the HOP rewards matrix, a learning rate of 0.1, a discount rate of 0.8, and a target state of S7 (the location to the far right of the wooden board). I did 10,000 episodes (where in each one, the robot dropped randomly onto the wooden board and has to get to S7):

```
r.hop <- qlearn(HOP,10000,alpha=0.1,gamma=0.8,tgt.state=7)
> r.hop
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    0   51   64    0    0    0    0
[2,]    0    0   64    0    0    0    0
[3,]    0   51    0    0   80    0    0
[4,]    0    0   64    0   80   80    0
[5,]    0    0    0    0    0   80  100
[6,]    0    0    0    0   80    0  100
[7,]    0    0    0    0    0   80  100
```

The Q-Matrix that is presented encodes the best-value solutions from each state (the "policy"). Here's how you read it:

- If you're at s1 (first row), **hop** to s3 (biggest value in first row), then hop to s5 (go to row 3 and find biggest value), then hop to s7 (go to row 5 and find biggest value)
- If you're at s2, go **right** to s3, then hop to s5, then hop to s7
- If you're at s3, **hop** to s5, then hop to s7
- If you're at s4, go **right** to s5 OR hop to s6, then go right to s7
- If you're at s5, **hop** to s7
- If you're at s6, go **right** to s7
- If you're at s7, **stay there** (when you're in the target state, the value function will not be able to pick out a "best action" because the best action is to do nothing)

Alternatively, the policy can be expressed as the best action from each of the 7 states: **HOP, RIGHT, HOP, RIGHT, HOP, RIGHT, (STAY PUT)**

Attempt 2: Use ReinforcementLearning Package

I also used the ReinforcementLearning package by Nicholas Proellochs (6/19/2017) described in https://cran.r-project.org/web/packages/ReinforcementLearning/ReinforcementLearning.pdf (https://cran.r-project.org/web/packages/ReinforcementLearning/ReinforcementLearning.pdf).

- **Input**: 1) a definition of the environment, 2) a list of states, 3) a list of actions, and 4) control parameters **alpha** (the learning rate; usually 0.1), **gamma** (the discount rate which describes how important future rewards are; often 0.9 indicating that 90% of the next reward will be taken into account), and **epsilon** (the probability that you'll try a random action; often 0.1)
- **Output**: A State-Action Value matrix, which attaches a number to how good it is to be in a particular state *and* take an action. You can use it to determine the highest value action from each state. (It contains the same information as the Q-matrix from Attempt 1, but you don't have to infer the action from the destination it brings you to.)
- **Pros**: Relatively straightforward. Allows you to specify epsilon, which controls the proportion of *random actions* you'll explore as you create episodes and explore your environment. **Cons**: Requires manual setup of all state transitions and associated rewards.

First, I created an "environment" that describes 1) how the states will change when actions are taken, and 2) what rewards will be accrued when that happens. I assigned a reward of -1 to all actions that are not special, e.g. landing on S1, landing on S4, or landing on S7. To be perfectly consistent with Attempt 1, I could have used 0.01 instead of -1, but the results will be similar. The values you choose for rewards are sort of arbitrary, but you do need to make sure there's a comparatively large positive reward at your target state and "negative rewards" for states you want to avoid or are physically impossible.

```
my.env <- function(state,action) {
   next_state <- state
   if (state == state("s1") && action == "right")  { next_state
   if (state == state("s1") && action == "hop")    { next_state

   if (state == state("s2") && action == "left")  {
    next_state <- state("s1"); reward <- -10 }
   if (state == state("s2") && action == "right") { next_state <
   if (state == state("s2") && action == "hop")   {
    next_state <- state("s4"); reward <- -3 }

   if (state == state("s3") && action == "left")  { next_state <
   if (state == state("s3") && action == "right") {
    next_state <- state("s4"); reward <- -3 }
   if (state == state("s3") && action == "hop")   { next_state <

   if (state == state("s4") && action == "left")  { next_state <
   if (state == state("s4") && action == "right") { next_state <
   if (state == state("s4") && action == "hop")   { next_state <
```

```
    if (state == state("s5") && action == "left")  {
     next_state <- state("s4"); reward <- -3 }
    if (state == state("s5") && action == "right") { next_state <
    if (state == state("s5") && action == "hop")   {
     next_state <- state("s7"); reward <- 10 }

    if (state == state("s6") && action == "left")  { next_state <
    if (state == state("s6") && action == "right") {
     next_state <- state("s7"); reward <- 10 }

    if (next_state == state("s7") && state != state("s7")) {
         reward <- 10
    } else {
     reward <- -1
    }
    out <- list(NextState = next_state, Reward = reward)
    return(out)
}
```

Next, I installed and loaded up the ReinforcementLearning package and ran the RL simulation:

```
install.packages("ReinforcementLearning")
library(ReinforcementLearning)
states <- c("s1", "s2", "s3", "s4", "s5", "s6", "s7")
actions <- c("left","right","hop")
data <- sampleExperience(N=3000,env=my.env,states=states,actions=a
control <- list(alpha = 0.1, gamma = 0.8, epsilon = 0.1)
model <- ReinforcementLearning(data, s = "State", a = "Action", r
      s_new = "NextState", control = control)
```

Now we can see the results:

```
> print(model)
State-Action function Q
         hop     right      left
s1  2.456741  1.022440  1.035193
s2  2.441032  2.452331  1.054154
s3  4.233166  2.469494  1.048073
s4  4.179853  4.221801  2.422842
s5  6.397159  4.175642  2.456108
s6  4.217752  6.410110  4.223972
s7 -4.602003 -4.593739 -4.591626

Policy
     s1      s2      s3      s4      s5      s6      s7
  "hop" "right"   "hop" "right"   "hop" "right"  "left"

Reward (last iteration)
[1] 223
```

The recommended policy is: **HOP, RIGHT, HOP, RIGHT, HOP, RIGHT, (STAY PUT)**

If you tried this example and it didn't produce the same response, don't worry! Model-free reinforcement learning is done by simulation, and when you used the sampleExperience function, you generated a different set of state transitions to learn from. You may need more samples, or to tweak your rewards structure, or both.)

5 replies »

Pingback: Reinforcement Learning: Q-Learning with the Hopping Robot – Mubashir Qasim

Hi, Reply

Looks like there's a tiny typo in your R code (probalby copy-paste errors).

In attempt 2: "y.env" probalby is "my.env"

Great article.

> Thank you! WordPress seems to chomp at least three parts of my code e Reply
> time I publish a post. I caught the first two, so thanks for catching the thir

Pingback: Distilled News | Data Analytics & R

Thank you for your good example. I learn a lot more clearer about RL now. Reply

Just in the Attempt 2 my.env function, it seems that the last "if" clause will overwrite the -3 penalty for state 4.
As a result, RL sometimes gets the results of good policy as "hop" for state 2 into state 4 without penalty faster towards state 7.
If changing s4 as -3 for the data, RL can get the correct results.

Thank you.