Week 5: Recursion in programming, cognition and networks

Willem Zuidema, Phong Le

November 27, 2013

1 Functions

Over the last few weeks we have seen examples of functions that are built-in in R, as well as a number of scripts that defined new functions. Functions are the building blocks of computer programs - the bits of code that we give a separate name so that we can reuse them again and again. Today, we are going to define some new functions ourselves.

The structure of a function is given as follows

```
1 func_name <- function(arg1, arg2, ...) {
2     statement
3     return(output)
4 }
```

where `func_name` is the function's name; `arg1`, `arg2`, etc. are the function's arguments (i.e., function's inputs); and `return(output)` is to claim that the output of the function is the object `output`, which need to be declared in the body of the function (i.e., `statement`).

Exercise 1.1: Write a function `sigmoid` that, for a given x calculates the y value. Recall that the sigmoid function is defined as:

$$y = \frac{1}{1 + e^{-x}}$$

Check that the function works equally well with x a scalar (a single value of x) as with x a vector. Plot the function with:

```
1 x <- seq(-5,5,by=0.1)          # create x = (-5,-4.9,-4.8,...,4.9,5)
2 y <- sigmoid(x)                    # compute y values
3 plot(x,y,type='l',col='blue')  # draw the graph, using blue color
```

2 Scripts

Programming directly in an R console is a bad idea because we can't bring our programs to another computer. Therefore, we should store our source code in a file and execute it anytime we want by the function `source("file_path")`.

Exercise 2.1: Store your code in Exercise 1.1 in a file "sigmoid.r" and execute it by typing `source("sigmoid.r")` in an R console. (If the error "cannot open file" occurs, you change your working directory.)

3 Control Structures

3.1 Conditional Execution

Think about what would happen if our languages lack the words 'if', 'else', 'otherwise', etc. and you will see why any programming language has to have conditional execution. In R, conditional execution is given by the following structure

```
if (condition) {
    statement
} else {
    alternative
}
```

The *condition* is a logical expression which may contain one of the following operations

- $x == y$ "x is equal to y"
- $x! = y$ "x is not equal to y"
- $x > y$ "x is greater than y"
- $x < y$ "x is less than y"
- $x <= y$ "x is less than or equal to y"
- $x >= y$ "x is greater than or equal to y"

or a combination of those using the & or && operators for AND, | or || are the operators for OR.

For instance, the following code will let us know, when we flip two fair coins, whether the results are both 'head'

```
x <- runif(2)   # flip two fair coins
print(x)
# suppose that "> 0.5" means "head"
# we check if BOTH two values > 0.5 or not
if (x[1] > 0.5 && x[2] > 0.5) {
    # both > 0.5
    print("bingo!!!")
} else {
    # at least one of them <= 0.5
    print("boooo!!!")
}
```

(Note: the "else" part is not necessary, you can omit it freely if you have nothing to do with it.)

> Exercise 3.1: Modify the code above to check whether at least one result is 'head'.

3.2 Loops

Suppose that we have a fair coin, and we flip it 1000 times and count how many times the coin turns head. To do that, we use the `for` statement like this way:

```
x <- runif(1000)   # flip a fair coin 1000 times
count <- 0
# suppose that "> 0.5" means "head"
# note that 1:1000 creates an array (1,2,3,...,1000)
for (i in 1:1000) {
    if (x[i] > 0.5) {
        count <- count + 1
    }
}
print(count)
```

Code 1: Using the `for` loop to count how many times a fain coin turns head.

The `for` loop above iterates each element in the array (1,2,...,1000). Firstly, the first element is assigned to i, and hence we check the value $x[1]$: if $x[1] > 0.5$ then we increase the *count* by 1. Then, the second element is assigned to i, and hence we check the value $x[2]$. And so on until the last element is assigned to i, and hence we check the value $x[10000]$.

We can wrap Code 1 into a function like this

```
# function count_head
# input: num_of_flips is a number of times we flip the coin
# output: the number of times the coin turns head
count_head <- function(num_of_flips) {
    x <- runif(num_of_flips)   # flip a fair coin
    count <- 0
    for (i in 1:num_of_flips) {
        if (x[i] > 0.5) {
            count <- count + 1
        }
    }
    return(count)
}
```

Now, to flip a fair coin 10000 times and count how many times it turns head, we simply execute `count <- count_head(10000)` (instead of typing Code 1 again and again each time we need to perform the task).

4 Recursion

R also allows us to write recursive functions: functions that call themselves! This can be extremely useful, but should be handled with care. Recursive functions always have (at least) two parts: a stop-condition (to avoid infinite loops) and a recursive step. As an easy example, consider a function `mysum` that sums all elements from row $r1$ until row $r2$ in a vector x (of course, we could easily do that with the built-in function `sum()`, or write our own function with a for-loop, but a recursive solution is more fun):

$$mysum(x, r1, r2) = \begin{cases} x[r1] & if(r1 == r2) \\ x[r1] + mysum(x, r1 + 1, r2) & otherwise \end{cases}$$

> Exercise 4.1: Implement this function in R, and test it on some example vectors.

We can use recursive function also to generate strings, using the framework of formal grammars. For instance, we can generate the formal language $(ab)^n$ with the function below (that uses the built-in function `cat` to print comments on the screen, and the built-in function `paste(a,b,c,...)` to glue subtrings a, b, c, etc. together into one string):

```
abn <- function(n) {
  cat(paste("entering abn with n=",n,"\n",sep=""))

# STOP CONDITION
  if (n==0) {
    result <- ""
  }
# RECURSIVE STEP
  else {
    intermediate_result <- abn(n-1)
    result <- paste(intermediate_result,"ab",sep="")
  }
# RETURN RESULT
  cat(paste("leaving abn with n=",n,", result=",result,"\n",sep=""))
  return(result)
}
```

Exercise 4.2: Generate some string with this function for randomly chosen n's. Make sure you understand the messages on your screen about entering and leaving abn with a particular n.

Exercise 4.3: Write a recursive function to generate strings from $a^n b^n$.

5 Recurrent networks

Recall from previous weeks that the input to a McCulloch-Pitts neuron is the weighted sum of the activations of input neurons, which we can succinctly describe as vector multiplication. More generally, if we have a network of neurons, we can describe their activations with one activation vector $\mathbf{x}$, the weights between all neurons with a weight matrix $\mathbf{W}$, and the net input to all neurons with a vector $\mathbf{y} = \mathbf{x} \cdot \mathbf{W}$. When two neurons i and j are not connected, the corresponding entry in the weight matrix $\mathbf{W}_{ij}$ is simply 0.

To calculate the new activation vector $\mathbf{x}_{t+1}$ given the current $\mathbf{x}_t$ and the weights $\mathbf{W}$, all we have to do is apply our threshold function of choice:

$$\mathbf{x}_{t+1} = \text{threshold}(\mathbf{x} \cdot \mathbf{W})$$

For instance, if we use as threshold function the sign-function, we can describe a network that implements XOR with the following weight matrix (where i_1 and i_2 are the two neurons that form the input layer, h_1 and h_2 are the two neurons that form the hidden layer, o the output neuron, and b the bias neuron).

$$x = \left(\begin{array}{c|c} i_1 & 1 \\ i_2 & 0 \\ h_1 & 0 \\ h_2 & 0 \\ o & 0 \\ b & 1 \end{array}\right) \mathbf{W} = \left(\begin{array}{c|cccccc} & i_1 & i_2 & h_1 & h_2 & o & b \\ \hline i_1 & 0 & 0 & 1 & -1 & 0 & 0 \\ i_2 & 0 & 0 & 1 & -1 & 0 & 0 \\ h_1 & 0 & 0 & 0 & 0 & 1 & 0 \\ h_2 & 0 & 0 & 0 & 0 & 1 & 0 \\ o & 0 & 0 & 0 & 0 & 0 & 0 \\ b & 0 & 0 & -0.5 & 1.5 & -1.5 & 100 \end{array}\right)$$

Recall that you can enter a matrix in R as follows:

```
W <- matrix(c(0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,-0.5,-1,
-1,0,0,0,1.5,0,0,1,1,0,-1.5,0,0,0,0,0,100),6,6)
```

and do matrix multiplication with `%*%`.

You can define the threshold function using

```
threshold <- function(x) { (x>0)+0 }
```

Exercise 5.1: Test whether this weight matrix indeed implements XOR by trying out all four relevant input values (00,01,10,11) in the first two entries of vector $\mathbf{x}$. Hint: the output node activation is in the 5th entry of $\mathbf{x}$, and to compute its value you need to compute $\mathbf{x}_{t+2}$ (i.e., do the matrix-multiplication and thresholding twice).

Exercise 5.2: Write a recursive function with an argument T that computes the vector $\mathbf{x}_t$ at time $t = T$.

In the example weight matrix, the weights from each layer to itself are all zero. We can easily make the network *recurrent* by making these weights non-zero. If we allow nonzero weights between all hidden layer nodes we have a simple recurrent network, which has been calimed to be able to implement a contextfree

grammar. We cannot investigate that claim in detail here, but to sharpen our intuitions let's try to design a simple recurrent network that processes strings like aabb, aaabbb, aaaaabbbbb etc. by receiving one character at a time and predicting the next character. For this, we need a network with two input neuron and two output neurons (where we let 00 mean "beginning of the string", 01 'a', 10 'b' and 11 "end of string").

> Exercise 5.2: Implement a recurrent network with 2 input, 2 hidden and 2 output neurons that processes "aaabbb". Encode the strings as a matrix, and write a function that presents every row of the matrix to the network at each timestep.

6 Towers of Hanoi

Now that you understand recursive programming, we can analyze the optimal solution to Towers of Hanoi problems. First load the package `ref`

```
install.packages("ref")
require("ref")
```

Set some variables:

```
n=4
Tower <- list(Tower = list(a = n:1, b = numeric(), c = numeric()))
class(Tower) <- "HanoiTower"
```

Then define the function to move a tower:

```
move.HTower <- function (Tower, print=FALSE, plot=TRUE, n = 1, from = 1, to = 1)
    {

    if (n == 1) {
        nfrom <- length(Tower$Tower[[from]])
        Tower$Tower[[to]][length(Tower$Tower[[to]]) + 1] <- Tower$Tower[[from]][
            nfrom]

        length(Tower$Tower[[from]]) <- nfrom - 1
        if (print) print.HanoiTower(Tower)
        if (plot) plot.HanoiTower(Tower)
    }
    else {
        free <- (1:3)[-c(from, to)]
        Tower <- move.HTower(Tower, print, plot, n - 1, from, free)
        Tower <- move.HTower(Tower, print, plot, 1, from, to)
        Tower <- move.HTower(Tower, print, plot, n - 1, free, to)
    }

    Sys.sleep(0.5) #readline()
    Tower
}
```

Experiment with different moves by executing e.g.:

```
move.HTower(Tower, n=n, from = 1, to = 2)
```

> Exercise 6.1: Explain how the function move.HTower works.

7 Submission

You have to submit a file named 'your_name.pdf'.

A More control structures

Suppose we want to flip the coin until it turns head. We can't use the `for` statement because we don't know how many times we need to flip the coin. Here, we need another loop statement, namely `while`:

```
while (condition) {
    statement
}
```

The `while` loop will execute the `statement` as long as the `condition` is correct (i.e., `TRUE`). Therefore, our problem is solved by the following code

```
# suppose that "> 0.5" means "head"
x <- runif(1)   # flip the coin
print(x)
while (x <= 0.5) {
    x <- runif(1)   # flip the coin
    print(x)
}
print("bingo!!!")
```

B Indexing

We can get an element by using the operator `[]`. For instance, `vec[i]` points to the i-th element of a vector `vec`, `mat[i,j]` points to the element on the i-th row, the j-th column of a matrix `mat`.

The operator `[]` can do further than that: we can get a set of elements. For instance, `a[c(1,3)]` and `a[c(TRUE,FALSE,TRUE)]` point to the first and the third elements of vector `a`; `A[1,c(2,3)]` and `A[1,c(FALSE,TRUE,TRUE)]` point to the second and the third elements on the first row of matrix `A`, and `A[,2]` points to the second column.