`rnn`: a Recurrent Neural Network in R*

Bastiaan Quast†

12th April 2016

Abstract

The rnn package implements a Recurrent Neural Network (RNN). RNN algorithms have the ability to train neural networks to deal with greater levels of complexity . This package is purposely designed to demonstrate the self learning ability using the classic example of binary summation on a bit-by-bit (right to left) basis, which requires the model to develop the understanding that if a 1 and 1 one are added, the outcome is 0, but in the next iteration, it has to remember to carry 1.

*https://cran.r-project.org/package=rnn | https://github.com/bquast/rnn

†http://qua.st | bastiaan.quast@graduateinstitute.ch | bquast@gmail.com

1 Introduction

This package implements a neural network which is used to convert numbers in range of 0-127 to binary representation.

Of course, numbers < 128 can be represent in a 7-bit binary form, but since we are adding two numbers in the range 0-127, the total can reach and achieve 128, which requires 8 bits, it cannot be more than 254, the limit of 8 bit binary representation is 255, thereby preventing overflows.

At this point it is useful to clarrify the nomenclature in this article. I use the term RNN (capitalised) for the general concept of a Recurrent Neural Network. Other than than I use `rnn` (in miniscules) to refer to the R package, lastly I use `rnn()` (in miniscules with brackets as a suffix) to refer to the central function of the `rnn` package, which is an implementation of a Recurrent Neural Network (RNN).

```
# load the package
library(rnn)

# list functions
ls('package:rnn')

## [1] "int2binary"                    "rnn"
## [3] "sigmoid"                       "sigmoid_output_to_derivative"
```

2 Data

The main rnn() function takes three integer vectors as inputs: Y, X1, and X2. The vectors X1 and X2 are independent variables, the Y vector is the sum of X1 and X2 and is the response variable (for more info see `help('rnn')`).

```
# use the same random numbers
set.seed(123)

# create training inputs
X1 = sample(0:127, 7000, replace=TRUE)
X2 = sample(0:127, 7000, replace=TRUE)

# create training output
Y <- X1 + X2

# check that all vectors are integer
typeof(c(X1,X2,Y))

## [1] "integer"
```

Internally the `rnn()` function converts these characters into binary format using the `int2binary()` function.

We can for instance take the first value of `X1` and convert it to a binary representation, whereby the length argument to the function determines the length of the binary representation, throughout this paper we will use 8 bit representations (which limits numbers to the range 0-255), but the program is by no mean limited to that.

```
X1[1]

## [1] 36

int2binary(X1[1], length=8)

## [1] 0 0 1 0 0 1 0 0
```

Lets check look at the first sum in decimal representation.

```
X1[1]
X2[1]
X1[1] + X2[1]
Y[1]

## [1] 36
## [1] 119
## [1] 155
## [1] 155
```

and now in binary representation.

```
int2binary(X1[1], length=8)
int2binary(X2[1], length=8)
print('--------------')
int2binary(Y[1],  length=8)

## [1] 0 0 1 0 0 1 0 0
## [1] 0 1 1 1 0 1 1 1
## [1] "--------------"
## [1] 1 0 0 1 1 0 1 1
```

As can be seen from the above output, the first values of `X1` and `X2`, `36` and `119` respectively, are both in the range 0-127, which can be represented with only 7 bits. Yet the sum of the two - 155 - is outside of the range 0-127, which is why an 8th bit is required (i.e. the 8th value from right to left in the bottom row is `1`). If we sampled numbers great than 127 for `X1` and `X2` then the sum of the two could be greater than 255, which requires a ninth bit (or `length=9`)

The rnn() function will run until it has evaluated all values in the vector that it is fed. Since the training of the network, particularly the carrying part, takes

many iterations to learn (the exact number of iterations varies but depends on the hyperparameters, more on this in the next section), it is therefore advisable to sample several thousand values (I use 7000).

3 Methodology

The workhorese of the `rnn` package is the `rnn()` function.

If we go back to the output of the int2binary() function for X1, X2, and Y, we see that in the 4th column (from right to left), a 0 and a 0 are added, resulting in an output of 1. This is because in the previous iteration 3rd column (from right to left) a 1 and a 1 are added, which becomes 1 0, so the 0 goes in column 3 and the 1 is carried over to colum 4. Since the summation is done bit by bit (or column by column), the neural network need to remember from the 3rd iteration until the 4th interation that it is carrying a 1 over. It is this remembering that a feed-forward neural network cannot teach itself.

The rnn() function internally makes use of the `sigmoid()` function, which is a very simple implementation of a sigmoid which takes the range (-Infinity, Infinity) and maps it to the range (0, 1).

```
# print source code of the sigmoid function
sigmoid

## function(x) {
##   output = 1 / (1+exp(-x))
##   return(output)            }
## <environment: namespace:rnn>
```

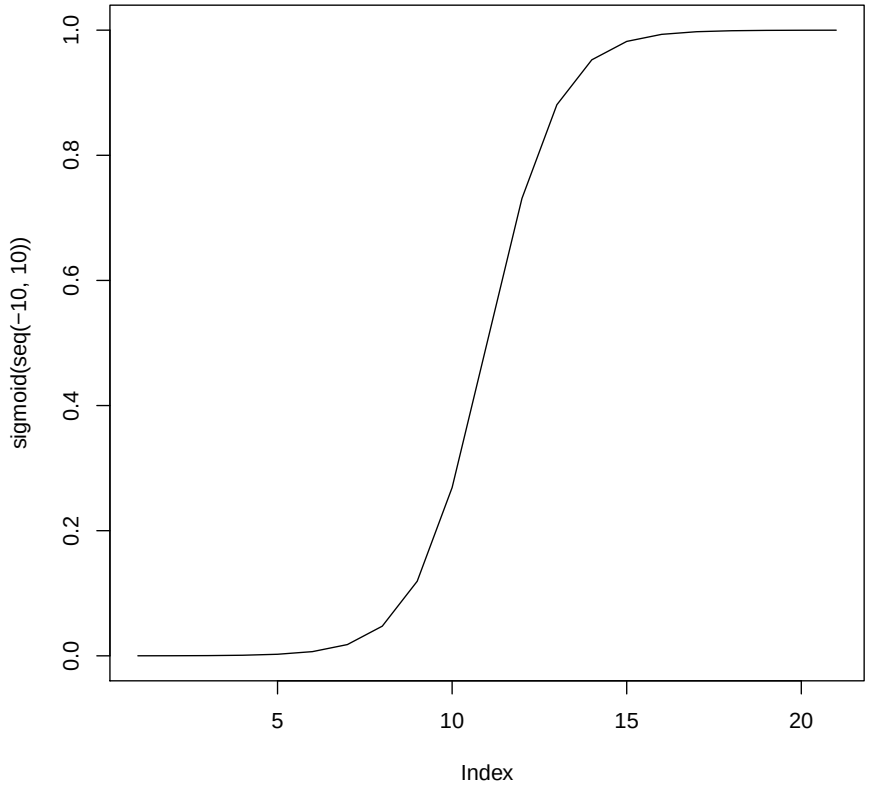
For instance:

```
sigmoid(-137)
sigmoid(5.3)

## [1] 3.174359e-60
## [1] 0.9950332
```

The shape of the sigmoid function is as follows.

```
plot(sigmoid(seq(-10,10)), type='l')
```



Additionally the rnn() function uses the sigmoid_output_to_derivative() function.

```
# print source code of the sigmoid_output_to_derivate function
sigmoid_output_to_derivative

## function(output) {
##   return( output*(1-output) )                    }
## <environment: namespace:rnn>
```

4 Results

As the purpose of the package is to illustrate the working of a Recurrent Neural Network, the `rnn()` function is quite verbose (this can be controlled using the `print` argument).

```
## [1] "Summation number: 1000"
## [1] "x1: 1"
## [1] "x2: 1 +"
## [1] "-------"
## [1] "y:  0"
## [1] "y^: 0"
## [1] "======="
## [1] "x1: 0"
## [1] "x2: 0 +"
## [1] "-------"
## [1] "y:  1"
## [1] "y^: 1"
## [1] "======="
## [1] "x1: 1"
## [1] "x2: 0 +"
## [1] "-------"
## [1] "y:  1"
## [1] "y^: 0"
## [1] "======="
## [1] "x1: 1"
## [1] "x2: 1 +"
## [1] "-------"
## [1] "y:  0"
## [1] "y^: 1"
## [1] "======="
## [1] "x1: 0"
## [1] "x2: 1 +"
## [1] "-------"
## [1] "y:  0"
## [1] "y^: 1"
## [1] "======="
## [1] "x1: 0"
## [1] "x2: 1 +"
## [1] "-------"
## [1] "y:  0"
## [1] "y^: 1"
## [1] "======="
## [1] "x1: 0"
## [1] "x2: 0 +"
## [1] "-------"
## [1] "y:  1"
## [1] "y^: 0"
## [1] "======="
## [1] "x1: 0"
## [1] "x2: 0 +"
```

```
## [1] "-------"
## [1] "y:  0"
## [1] "y^: 0"
## [1] "======="
## [1] "Error: 4.09583204978584"
## [1] "X1[ 1000 ]: 0 0 0 0 1 1 0 1   ( 13 )"
## [1] "X2[ 1000 ]: 0 0 1 1 1 0 0 1 + ( 57 )"
## [1] "-----------------------------"
## [1] "Y[ 1000 ]:  0 1 0 0 0 1 1 0   ( 70 )"
## [1] "predict Y^: 0 0 1 1 1 0 1 0   ( 58 )"
## [1] "============================="
```

The text printed here is of the 8 steps of the summation of the 1000th value of `X1` and `X2`, or iteration 7993-8000.

Each iteration is printed individually, with the two input bits, the prediction for the response value and the actual response value.

After each iternation the difference betweeen the predicted value and the actual value is fed back into the neural network using a method called back-propagation (an application the chain rule of differential calculus).

At the end of the 8 iterations that it here takes to add two values of `X1` and `X2`, the results are printed in a more human legible form. It should be clear from the results that after 1000 numbers, which 8 iterations each, the model is still performing very poorly.

However, progress can be seen:

```
# use the same random numbers
set.seed(123)

# train the network
rnn(Y,
    X1,
    X2,
    binary_dim =  8,
    alpha      =  0.1,
    input_dim  =  2,
    hidden_dim = 10,
    output_dim =  1,
    print = 'minimal' )

## [1] "Summation number: 1000"
## [1] "Error: 4.00320706549242"
## [1] "X1[ 1000 ]: 0 0 0 0 1 1 0 1   ( 13 )"
## [1] "X2[ 1000 ]: 0 0 1 0 1 1 1 1 + ( 47 )"
## [1] "-----------------------------"
## [1] "Y[ 1000 ]:  0 0 1 1 1 1 0 0   ( 60 )"
## [1] "predict Y^: 0 1 1 1 1 1 1 1   ( 127 )"
```

```
## [1] "============================="
## [1] "Summation number: 2000"
## [1] "Error: 4.23764606054627"
## [1] "X1[ 2000 ]: 0 0 1 1 1 0 0 1   ( 57 )"
## [1] "X2[ 2000 ]: 0 1 1 0 0 1 0 0 + ( 100 )"
## [1] "-----------------------------"
## [1] "Y[ 2000 ]:  1 0 0 1 1 1 0 1   ( 157 )"
## [1] "predict Y^: 0 1 1 0 0 1 1 0   ( 102 )"
## [1] "============================="
## [1] "Summation number: 3000"
## [1] "Error: 3.87343759636473"
## [1] "X1[ 3000 ]: 0 1 0 1 0 1 1 1   ( 87 )"
## [1] "X2[ 3000 ]: 0 1 1 1 0 0 1 1 + ( 115 )"
## [1] "-----------------------------"
## [1] "Y[ 3000 ]:  1 1 0 0 1 0 1 0   ( 202 )"
## [1] "predict Y^: 0 0 0 0 0 0 0 0   ( 0 )"
## [1] "============================="
## [1] "Summation number: 4000"
## [1] "Error: 3.58000119107544"
## [1] "X1[ 4000 ]: 0 0 1 1 0 1 1 0   ( 54 )"
## [1] "X2[ 4000 ]: 0 0 1 1 1 0 0 0 + ( 56 )"
## [1] "-----------------------------"
## [1] "Y[ 4000 ]:  0 1 1 0 1 1 1 0   ( 110 )"
## [1] "predict Y^: 0 1 0 0 0 0 0 0   ( 64 )"
## [1] "============================="
## [1] "Summation number: 5000"
## [1] "Error: 2.87293795267989"
## [1] "X1[ 5000 ]: 0 0 0 0 0 0 0 1   ( 1 )"
## [1] "X2[ 5000 ]: 0 0 1 1 1 0 1 0 + ( 58 )"
## [1] "-----------------------------"
## [1] "Y[ 5000 ]:  0 0 1 1 1 0 1 1   ( 59 )"
## [1] "predict Y^: 0 0 1 1 1 0 1 1   ( 59 )"
## [1] "============================="
## [1] "Summation number: 6000"
## [1] "Error: 1.47531055189879"
## [1] "X1[ 6000 ]: 0 1 1 1 0 0 1 1   ( 115 )"
## [1] "X2[ 6000 ]: 0 0 1 0 0 1 0 1 + ( 37 )"
## [1] "-----------------------------"
## [1] "Y[ 6000 ]:  1 0 0 1 1 0 0 0   ( 152 )"
## [1] "predict Y^: 1 0 0 1 1 0 0 0   ( 152 )"
## [1] "============================="
## [1] "Summation number: 7000"
## [1] "Error: 0.980964833440344"
## [1] "X1[ 7000 ]: 0 1 0 1 0 1 1 1   ( 87 )"
## [1] "X2[ 7000 ]: 0 0 1 1 1 0 1 1 + ( 59 )"
```

```
## [1] "-----------------------------"
## [1] "Y[ 7000 ]:  1 0 0 1 0 0 1 0   ( 146 )"
## [1] "predict Y^: 1 0 0 1 0 0 1 0   ( 146 )"
## [1] "=============================="
```

In fact, from the 5000th summation on, all the printed estimates are in fact correct.

5 Conclusion

CRAN and the rest of the R ecosystem show that there is a strong interest in using the R language for neural network analysis. Existing package such as the built in nnet package and the caret package make available very powerful neural network tools to R users. The RSNNS package acts as an R wrapper for the Stutgard Neural Network Simulator library, which is written in C, and thereby makes available to partial RNNs such as Elman and Jordan networks.

The enormous popularity of full Recurrent Neural Networks in other languages, primarily Python, show that there is a great amount of interest for using this methodology, including interest from Economist, Data Scientists, and other non-professional programmers. Although Python is a relatively accessible programming language for laymen, it has a smaller user base in terms of data analists. The `rnn` package attempts to address this need by showing that Recurrent Neural Networks can be made available and perhaps more importantly, made available in native R, which allows user to delve into the code and understand the method and developer a more thorough understanding of how to use it.

A Source code of `rnn()`

```
rnn

## function(Y, X1, X2, binary_dim, alpha, input_dim, hidden_dim, output_dim, print = c('full
##
##   # check what largest possible number is
##   largest_number = 2^binary_dim
##
##   # initialize neural network weights
##   synapse_0 = matrix(stats::runif(n = input_dim*hidden_dim, min=-1, max=1), nrow=input_di
##   synapse_1 = matrix(stats::runif(n = hidden_dim*output_dim, min=-1, max=1), nrow=hidden_
##   synapse_h = matrix(stats::runif(n = hidden_dim*hidden_dim, min=-1, max=1), nrow=hidden_
##
##   synapse_0_update = matrix(0, nrow = input_dim, ncol = hidden_dim)
##   synapse_1_update = matrix(0, nrow = hidden_dim, ncol = output_dim)
##   synapse_h_update = matrix(0, nrow = hidden_dim, ncol = hidden_dim)
```

```
##
##   # training logic
##   for (j in 1:length(Y)) {
##
##     if(print != 'none' && j %% 1000 == 0) {
##       print(paste('Summation number:', j))
##     }
##
##     # generate a simple addition problem (a + b = c)
##     a_int = X1[j] # int version
##     a = int2binary(a_int, binary_dim) # binary encoding
##
##     b_int = X2[j] # int version
##     b = int2binary(b_int, binary_dim)
##
##     # true answer
##     c_int = Y[j]
##     c = int2binary(c_int, binary_dim)
##
##     # where we'll store our best guesss (binary encoded)
##     d = matrix(0, nrow = 1, ncol = binary_dim)
##
##     overallError = 0
##
##     layer_2_deltas = matrix(0)
##     layer_1_values = matrix(0, nrow=1, ncol = hidden_dim)
##     # layer_1_values = rbind(layer_1_values, matrix(0, nrow=1, ncol=hidden_dim))
##
##     # moving along the positions in the binary encoding
##     for (position in 0:(binary_dim-1)) {
##
##       # generate input and output
##       X = cbind(a[binary_dim - position],b[binary_dim - position])
##       y = c[binary_dim - position]
##
##       # hidden layer (input ~+ prev_hidden)
##       layer_1 = sigmoid((X%*%synapse_0) + (layer_1_values[dim(layer_1_values)[1],] %*% sy
##
##       # output layer (new binary representation)
##       layer_2 = sigmoid(layer_1 %*% synapse_1)
##
##       # did we miss?... if so, by how much?
##       layer_2_error = y - layer_2
##       layer_2_deltas = rbind(layer_2_deltas, layer_2_error * sigmoid_output_to_derivative
##       overallError = overallError + abs(layer_2_error)
```

```
##
##       # decode estimate so we can print it out
##       d[binary_dim - position] = round(layer_2)
##
##       # store hidden layer so we can print it out
##       layer_1_values = rbind(layer_1_values, layer_1)
##
##       if(print == 'full' && j %% 1000 == 0) {
##         print(paste('x1:', a[binary_dim - position]))
##         print(paste('x2:', b[binary_dim - position], '+'))
##         print('-------')
##         print(paste('y: ', c[binary_dim - position]))
##         print(paste('y^:', d[binary_dim - position]))
##         print('=======')
##       }
##     }
##
##     future_layer_1_delta = matrix(0, nrow = 1, ncol = hidden_dim)
##
##     for (position in 0:(binary_dim-1)) {
##
##       X = cbind(a[position+1], b[position+1])
##       layer_1 = layer_1_values[dim(layer_1_values)[1]-position,]
##       prev_layer_1 = layer_1_values[dim(layer_1_values)[1]-(position+1),]
##
##       # error at output layer
##       layer_2_delta = layer_2_deltas[dim(layer_2_deltas)[1]-position,]
##       # error at hidden layer
##       layer_1_delta = (future_layer_1_delta %*% t(synapse_h) + layer_2_delta %*% t(synaps
##         sigmoid_output_to_derivative(layer_1)
##
##       # let's update all our weights so we can try again
##       synapse_1_update = synapse_1_update + matrix(layer_1) %*% layer_2_delta
##       synapse_h_update = synapse_h_update + matrix(prev_layer_1) %*% layer_1_delta
##       synapse_0_update = synapse_0_update + t(X) %*% layer_1_delta
##
##       future_layer_1_delta = layer_1_delta
##     }
##
##     synapse_0 = synapse_0 + ( synapse_0_update * alpha )
##     synapse_1 = synapse_1 + ( synapse_1_update * alpha )
##     synapse_h = synapse_h + ( synapse_h_update * alpha )
##
##     synapse_0_update = synapse_0_update * 0
##     synapse_1_update = synapse_1_update * 0
```

```
##     synapse_h_update = synapse_h_update * 0
##
##     # print out progress
##     if(print != 'none' && j %% 1000 == 0) {
##       print(paste('Error:', overallError))
##       print(paste('X1[', j, ']:', paste(a, collapse = ' '), ' ', '(', a_int, ')'))
##       print(paste('X2[', j, ']:', paste(b, collapse = ' '), '+', '(', b_int, ')'))
##       print('-----------------------------')
##       print(paste('Y[', j, ']: ', paste(c, collapse = ' '), ' ', '(', c_int, ')'))
##       out = 0
##       for (x in 1:length(d)) {
##         out[x] = rev(d)[x]*2^(x-1) }
##       print(paste('predict Y^:',   paste(d, collapse = ' '), ' ', '(', sum(out), ')'))
##       print('=============================')
##     }
##   }
## }
## <environment: namespace:rnn>
```