

rnn: a Recurrent Neural Network in R*

Bastiaan Quast[†]

14th 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 a 1 are added, the outcome is 0, but in the next iteration, it has to that it was carrying a 1 from the previous iteration.

*<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 Recurrent Neural Network which is trained to sum 8-bit binary numbers, teaching itself the complex task of carrying a 1 over to the next iteration if the sum of a column takes two bits of space.

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 and I use `rnn` (in miniscules and using a monospace font) to refer to the R package.

```
# load the package
library(rnn)

# list functions
ls('package:rnn')

## [1] "predictr"          "sigmoid"
## [3] "sigmoid_output_to_derivative" "trainr"
```

2 Data

The main `trainr()` 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('trainr')`).

```
# 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 `trainr()` function converts these characters into binary format using the `intToBits()` function and afterward converts it back into decimal format for printing using the `packBits()` function, both functions are included in the `base` package.

We can for instance take the first value of `X1` and convert it to a binary representation, whereby the `binary_dim` argument to the `trainr()` 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 theoretical limit is 32 bits.

```
# manually define binary_dim
binary_dim = 8

X1[1]

## [1] 36

rev(as.numeric(intToBits( X1[1] ))[1:binary_dim])

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

Lets check look at the first sum in decimal representation.

```
X1[1]

## [1] 36

X2[1]

## [1] 119

X1[1] + X2[1]

## [1] 155

Y[1]

## [1] 155
```

and now in binary representation.

```
rev(as.numeric(intToBits( X1[1] ))[1:binary_dim])
rev(as.numeric(intToBits( X2[1] ))[1:binary_dim])
print('-----')
rev(as.numeric(intToBits( Y[1] ))[1:binary_dim])

## [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 workhorse of the **rnn** package is the **trainr()** function.

For example, if we add the binary numbers 0 0 1 (decimal system: 1) and 1 0 1 (decimal system: 5), we start by adding the right column, 1 and 1 make 1 0 (similar to when 5 and 5 make 1 0 in the decimal system) , the 0 is stored in the right column, the 1 is carried over to the middle column and added with the two existing bits 0 and 0, to form 1, which is stored in the middle column. This time nothing is carried over and the left column sums 0 and 1 to make 1, which gives the outcome 1 1 0 (decimal system: 6).

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 column 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 iteration 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>
```

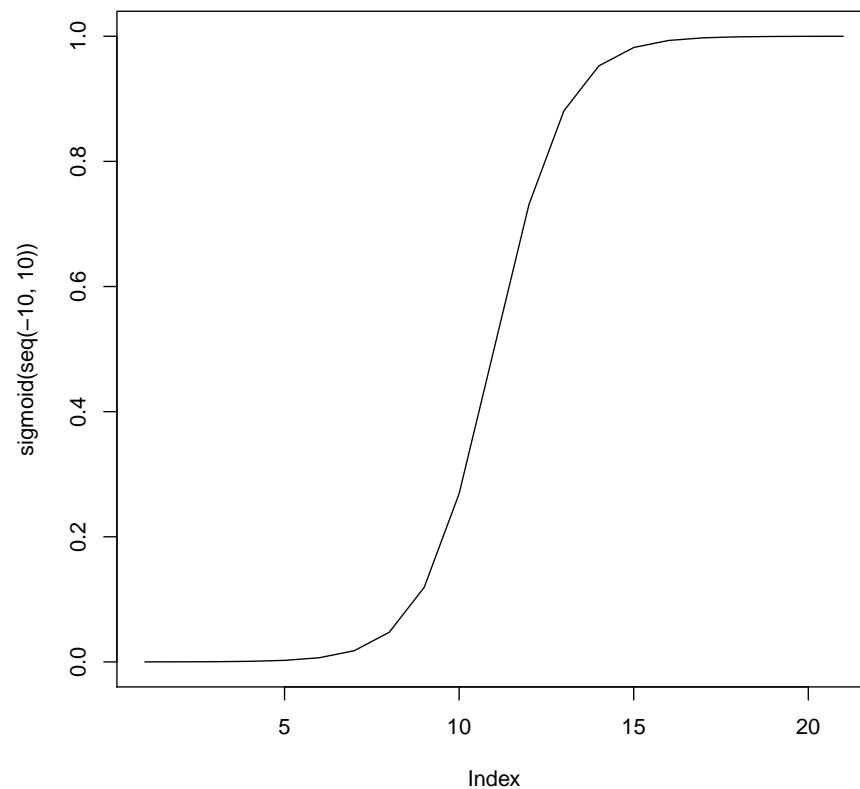
For instance:

```
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>
```

As the purpose of the package is to illustrate the working of a Recurrent Neural Network, the **trainr()** 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^: 1"
## [1] "======"
## [1] "x1: 0"
## [1] "x2: 0 +"
## [1] "-----"
## [1] "y:  1"
## [1] "y^: 1"
## [1] "======"
## [1] "x1: 0"
## [1] "x2: 0 +"
## [1] "-----"
## [1] "y:  0"
## [1] "y^: 0"
## [1] "======"
## [1] "x1: 0"
## [1] "x2: 0 +"
## [1] "-----"
## [1] "y:  0"
## [1] "y^: 0"
## [1] "======"
## [1] "x1: 0"
## [1] "x2: 1 +"
## [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: 0 +"
## [1] "-----"
## [1] "y:  1"
## [1] "y^: 1"
## [1] "======"
## [1] "x1: 0"
## [1] "x2: 0 +"
## [1] "-----"
## [1] "y:  0"
## [1] "y^: 0"
## [1] "======"
## [1] "Error: 3.87983649375435"
## [1] "X1[ 1000 ]: 0 0 1 0 0 0 0 1  ( 33 )"
## [1] "X2[ 1000 ]: 0 0 1 1 0 0 0 1 + ( 49 )"
## [1] "-----"
## [1] "Y[ 1000 ]:  0 1 0 1 0 0 1 0  ( 82 )"
## [1] "predict Y^: 0 1 1 0 0 0 1 1  ( 99 )"
## [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 iteration the difference between 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 observed:

```
# use the same random numbers
set.seed(1)

# train the network
m1 <- trainr(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: 3.90950698309064"
## [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: 4.03678792609062"
## [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~: 1 1 1 1 1 1 1 1 ( 255 )"
## [1] "=====
## [1] "Summation number: 3000"
## [1] "Error: 4.0117145610462"
## [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~: 1 0 0 0 0 1 0 0 ( 132 )"
## [1] "=====
## [1] "Summation number: 4000"
## [1] "Error: 3.4363646618886"
## [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 1 0 ( 66 )"
## [1] "=====
## [1] "Summation number: 5000"
## [1] "Error: 2.26331897903431"
## [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 1 0 1 ( 61 )"
## [1] "=====
## [1] "Summation number: 6000"
## [1] "Error: 1.90553916946373"
## [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: 0.992546604462261"
## [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 6000th summation on, all the printed estimates are in fact correct.

4 Results

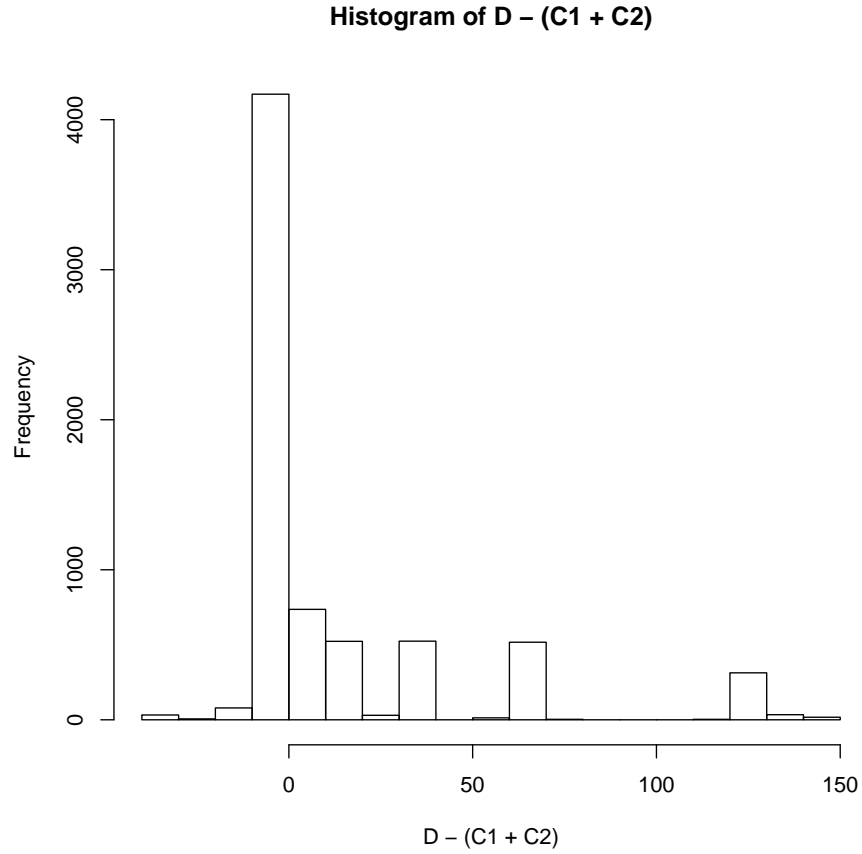
The eventual purpose is to use the model generated by the `trainr()` function as an input to the `predictr()` function, in order to predict the values of new inputs.

```
# create test inputs
C1 = sample(0:127, 7000, replace=TRUE)
C2 = sample(0:127, 7000, replace=TRUE)
```

Now predict using the `predictr()` function.

```
# predic
D <- predictr(m1,
               C1,
               C2,
               binary_dim = 8,
               alpha      = 0.1,
               input_dim  = 2,
               hidden_dim = 10,
               output_dim = 1 )

# inspect the differences
hist( D-(C1+C2) )
```



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 analysts. 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 trainr() function

1_Users_quast_rnn_R_trainr.R

```
1 #' @name trainr
2 #' @export
3 #' @importFrom stats runif
4 #' @title Recurrent Neural Network
5 #' @description Trains a Recurrent Neural Network.
6 #' @param Y vector of output values
7 #' @param X1 vector of input values
8 #' @param X2 vector of input values
9 #' @param binary_dim dimension of binary representation
10 #' @param alpha size of alpha
11 #' @param input_dim dimension of input layer, i.e. how many numbers
    to sum
12 #' @param hidden_dim dimension of hidden layer
13 #' @param output_dim dimension of output layer
14 #' @param print should train progress be printed
15 #' @return a model to be used by the predictr function
16 #' @examples
17 #' # create training inputs
18 #' X1 = sample(0:127, 7000, replace=TRUE)
19 #' X2 = sample(0:127, 7000, replace=TRUE)
20 #'
21 #' # create training output
22 #' Y <- X1 + X2
23 #'
24 #' # train the model
25 #' trainr(Y,
26 #'       X1,
27 #'       X2,
28 #'       binary_dim = 8,
29 #'       alpha      = 0.1,
30 #'       input_dim  = 2,
31 #'       hidden_dim = 10,
32 #'       output_dim = 1,
33 #'       print = 'full ' )
34 #'
35
36
37 trainr <- function(Y, X1, X2, binary_dim, alpha, input_dim, hidden_
    dim, output_dim, print = c('none', 'minimal', 'full')) {
38
39   # check what largest possible number is
40   largest_number = 2^binary_dim
41
42   # initialize neural network weights
43   synapse_0 = matrix(stats::runif(n = input_dim*hidden_dim, min=-1,
    max=1), nrow=input_dim)
44   synapse_l = matrix(stats::runif(n = hidden_dim*output_dim, min
    =-1, max=1), nrow=hidden_dim)
45   synapse_h = matrix(stats::runif(n = hidden_dim*hidden_dim, min
    =-1, max=1), nrow=hidden_dim)
46
47   synapse_0_update = matrix(0, nrow = input_dim, ncol = hidden_dim)
```

```

48 synapse_1_update = matrix(0, nrow = hidden_dim, ncol = output_dim
49 )
50 synapse_h_update = matrix(0, nrow = hidden_dim, ncol = hidden_dim
51 )
52 # training logic
53 for (j in 1:length(Y)) {
54     if(print != 'none' && j %% 1000 == 0) {
55         print(paste('Summation_number:', j))
56     }
57
58     # generate a simple addition problem (a + b = c)
59     a_int = X1[j] # int version
60     a = rev(as.numeric(intToBits(a_int))[1:binary_dim])
61
62     b_int = X2[j] # int version
63     b = rev(as.numeric(intToBits(b_int))[1:binary_dim])
64
65     # true answer
66     c_int = Y[j]
67     c = rev(as.numeric(intToBits(c_int))[1:binary_dim])
68
69     # where we'll store our best guess (binary encoded)
70     d = matrix(0, nrow = 1, ncol = binary_dim)
71
72     overallError = 0
73
74     layer_2_deltas = matrix(0)
75     layer_1_values = matrix(0, nrow=1, ncol = hidden_dim)
76     # layer_1_values = rbind(layer_1_values, matrix(0, nrow=1, ncol
77     # =hidden_dim))
78
79     # moving along the positions in the binary encoding
80     for (position in 0:(binary_dim-1)) {
81         # generate input and output
82         X = cbind(a[binary_dim - position], b[binary_dim - position])
83         y = c[binary_dim - position]
84
85         # hidden layer (input ~+ prev_hidden)
86         layer_1 = sigmoid((X%%synapse_0) + (layer_1_values[dim(layer
87         _1_values)[1],] %% synapse_h))
88
89         # output layer (new binary representation)
90         layer_2 = sigmoid(layer_1 %% synapse_1)
91
92         # did we miss?... if so, by how much?
93         layer_2_error = y - layer_2
94         layer_2_deltas = rbind(layer_2_deltas, layer_2_error *
95         sigmoid_output_to_derivative(layer_2))
96         overallError = overallError + abs(layer_2_error)
97
98         # decode estimate so we can print it out
99         d[binary_dim - position] = round(layer_2)
100
101         # store hidden layer so we can print it out

```

```

100     layer_1_values = rbind(layer_1_values, layer_1)
101
102     if(print == 'full' && j %% 1000 == 0) {
103         print(paste('x1:', a[binary_dim - position]))
104         print(paste('x2:', b[binary_dim - position], '+'))
105         print('_____')
106         print(paste('y:␣', c[binary_dim - position]))
107         print(paste('y^:', d[binary_dim - position]))
108         print('=====')
109     }
110 }
111
112 future_layer_1_delta = matrix(0, nrow = 1, ncol = hidden_dim)
113
114 for (position in 0:(binary_dim-1)) {
115
116     X = cbind(a[position+1], b[position+1])
117     layer_1 = layer_1_values[dim(layer_1_values)[1]-position,]
118     prev_layer_1 = layer_1_values[dim(layer_1_values)[1]-(
119         position+1),]
120
121     # error at output layer
122     layer_2_delta = layer_2_deltas[dim(layer_2_deltas)[1]-
123         position,]
124     # error at hidden layer
125     layer_1_delta = (future_layer_1_delta %*% t(synapse_h) +
126         layer_2_delta %*% t(synapse_1)) *
127         sigmoid_output_to_derivative(layer_1)
128
129     # let's update all our weights so we can try again
130     synapse_1_update = synapse_1_update + matrix(layer_1) %*%
131         layer_2_delta
132     synapse_h_update = synapse_h_update + matrix(prev_layer_1) %*
133         % layer_1_delta
134     synapse_0_update = synapse_0_update + t(X) %*% layer_1_delta
135
136     future_layer_1_delta = layer_1_delta
137 }
138
139 synapse_0 = synapse_0 + ( synapse_0_update * alpha )
140 synapse_1 = synapse_1 + ( synapse_1_update * alpha )
141 synapse_h = synapse_h + ( synapse_h_update * alpha )
142
143 synapse_0_update = synapse_0_update * 0
144 synapse_1_update = synapse_1_update * 0
145 synapse_h_update = synapse_h_update * 0
146
147 # print out progress
148 if(print != 'none' && j %% 1000 == 0) {
149     print(paste('Error:', overallError))
150     print(paste('X1[', j, ']:', paste(a, collapse = '␣'), '␣', '(
151         ', a_int, ')'))
152     print(paste('X2[', j, ']:', paste(b, collapse = '␣'), '+', '(
153         ', b_int, ')'))
154     print('_____')
155     print(paste('Y[', j, ']:␣', paste(c, collapse = '␣'), '␣', '(
156         ', c_int, ')'))

```

```

149     # convert d to decimal
150     out = packBits(as.raw(rev(c(rep(0, 32-binary_dim), d))), '
        integer')
151     print(paste('predict_Y^:', paste(d, collapse = '_'), '_ ', '
        ('', out, ' ')))
152     print('=====')
153 }
154 }
155
156 # output object with synapses
157 return(list(synapse_0 = synapse_0, synapse_1 = synapse_1, synapse
        _h = synapse_h))
158
159 }

```

B Source code of predictr() function

2_Users_quast_rnn_R_predictr.R

```

1 #' @name predictr
2 #' @export
3 #' @importFrom stats runif
4 #' @title Recurrent Neural Network
5 #' @description Trains a Recurrent Neural Network.
6 #' @param model output of the trainr function
7 #' @param X1 vector of input values
8 #' @param X2 vector of input values
9 #' @param binary_dim dimension of binary representation
10 #' @param alpha size of alpha
11 #' @param input_dim dimension of input layer, i.e. how many numbers
        to sum
12 #' @param hidden_dim dimension of hidden layer
13 #' @param output_dim dimension of output layer
14 #' @param print should train progress be printed
15 #' @return vector of predicted values
16 #' @examples
17 #' # create training inputs
18 #' X1 = sample(0:127, 7000, replace=TRUE)
19 #' X2 = sample(0:127, 7000, replace=TRUE)
20 #'
21 #' # create training output
22 #' Y <- X1 + X2
23 #'
24 #' # train the model
25 #' m1 <- trainr(Y,
26 #'             X1,
27 #'             X2,
28 #'             binary_dim = 8,
29 #'             alpha = 0.1,
30 #'             input_dim = 2,
31 #'             hidden_dim = 10,
32 #'             output_dim = 1 )
33 #'
34 #' # create test inputs
35 #' A1 = sample(0:127, 7000, replace=TRUE)

```

```

36 #' A2 = sample(0:127, 7000, replace=TRUE)
37 #'
38 #' # predict
39 #' B <- predictr(m1,
40 #'               A1,
41 #'               A2,
42 #'               binary_dim = 8,
43 #'               alpha      = 0.1,
44 #'               input_dim  = 2,
45 #'               hidden_dim = 10,
46 #'               output_dim = 1 )
47 #'
48 #' # inspect the differences
49 #' table( B-(A1+A2) )
50 #'
51
52
53 predictr <- function(model, X1, X2, binary_dim, alpha, input_dim,
54                       hidden_dim, output_dim, print = c('none', 'minimal', 'full')) {
55
56   # check what largest possible number is
57   largest_number = 2^binary_dim
58
59   # create output vector
60   Y <- vector(mode = 'integer', length = length(X1))
61
62   # load neural network weights
63   synapse_0 = model$synapse_0
64   synapse_1 = model$synapse_1
65   synapse_h = model$synapse_h
66
67   # synapse_0_update = matrix(0, nrow = input_dim, ncol = hidden_
68   # dim)
69   # synapse_1_update = matrix(0, nrow = hidden_dim, ncol = output_
70   # dim)
71   # synapse_h_update = matrix(0, nrow = hidden_dim, ncol = hidden_
72   # dim)
73
74   # training logic
75   for (j in 1:length(X1)) {
76
77     if(print != 'none' && j %% 1000 == 0) {
78       print(paste('Summation_number:', j))
79     }
80
81     # generate a simple addition problem (a + b = c)
82     a_int = X1[j] # int version
83     a = rev(as.numeric(intToBits(a_int)))[1:binary_dim]
84
85     b_int = X2[j] # int version
86     b = rev(as.numeric(intToBits(b_int)))[1:binary_dim]
87
88     # where we'll store our best guessss (binary encoded)
89     d = matrix(0, nrow = 1, ncol = binary_dim)
90
91     overallError = 0

```



```

89     layer_2_deltas = matrix(0)
90     layer_1_values = matrix(0, nrow=1, ncol = hidden_dim)
91     # layer_1_values = rbind(layer_1_values, matrix(0, nrow=1, ncol
92       =hidden_dim))
93
94     # moving along the positions in the binary encoding
95     for (position in 0:(binary_dim-1)) {
96
97         # generate input and output
98         X = cbind(a[binary_dim - position], b[binary_dim - position])
99
100        # hidden layer (input ~+ prev_hidden)
101        layer_1 = sigmoid((X%%synapse_0) + (layer_1_values[dim(layer_1_values)[1],] %% synapse_h))
102
103        # output layer (new binary representation)
104        layer_2 = sigmoid(layer_1 %% synapse_1)
105
106        # decode estimate so we can print it out
107        d[binary_dim - position] = round(layer_2)
108
109        # store hidden layer so we can print it out
110        layer_1_values = rbind(layer_1_values, layer_1)
111
112        if(print == 'full' && j %% 1000 == 0) {
113            print(paste('x1:', a[binary_dim - position]))
114            print(paste('x2:', b[binary_dim - position], '+'))
115            print('_____')
116            print(paste('y^:', d[binary_dim - position]))
117            print('=====')
118        }
119    }
120
121    # output to decimal
122    out = packBits(as.raw(rev(c(rep(0, 32-binary_dim), d))), '
123      integer')
124
125    # synapse_0 = synapse_0 + ( synapse_0_update * alpha )
126    # synapse_1 = synapse_1 + ( synapse_1_update * alpha )
127    # synapse_h = synapse_h + ( synapse_h_update * alpha )
128    #
129    # synapse_0_update = synapse_0_update * 0
130    # synapse_1_update = synapse_1_update * 0
131    # synapse_h_update = synapse_h_update * 0
132    #
133    # print out progress
134    if(print != 'none' && j %% 1000 == 0) {
135        print(paste('Error:', overallError))
136        print(paste('X1[', j, ']:', paste(a, collapse = '_'), '_', '(
137          ', a_int, ')'))
138        print(paste('X2[', j, ']:', paste(b, collapse = '_'), '+', '(
139          ', b_int, ')'))
140        print('_____')
141        # convert d to decimal

```

```

140     print(paste('predict_Y^:', paste(d, collapse = ' '), ' ', '
      ('', out, ' ')))
141     print('=====')
142 }
143
144     # store value
145     Y[j] <- out
146 }
147
148     # return output vector
149     return( Y )
150 }

```