

# `rnn`: a Recurrent Neural Network in R<sup>\*</sup>

Bastiaan Quast<sup>†</sup>

10th May 2016

## Abstract

The `rnn` package implements the first Recurrent Neural Network (RNN) in the R language. RNN algorithms expand on traditional feed-forward neural networks, allowing for greater complexity and dynamics, by implementing a memory state. This temporal nature of the algorithm makes it explicitly well suited for dynamic problems such as time series prediction. Additionally, this also allows for inputs of undefined or changing length, allowing models to be updated as new data comes in. The `rnn` package is the first implementation of a Recurrent Neural Network in the R language, making it both operable and understandable to R users. Here I apply the package to two problems, the classic complex problem of carrying a 1, in bit by bit (column by column) binary addition, as well as foreign exchange rate prediction. Interactive live versions of these examples are available on my website <http://qua.st/rnn>.

## 1 Introduction

This package implements the first Recurrent Neural Network in the R language.

At this point it is useful to clarify 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 minuscules and using a monospace font) to refer to the R package.

---

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

<sup>†</sup><http://qua.st> | [bastiaan.quast@graduateinstitute.ch](mailto:bastiaan.quast@graduateinstitute.ch) | [bquast@gmail.com](mailto:bquast@gmail.com)

Table 1: `rnn` Package

```
# load library
library(rnn)

# list included functions
ls('package:rnn')

## [1] "bin2int"          "int2bin"          "predictr"
## [4] "run.finance_demo" "trainr"
```

As is listed above, the package contains the following functions:

- `bin2int()`: conversion of a matrix of numbers in binary representation to decimal representation;
- `int2bin()`: conversion of a vector numbers in decimal representation to binary representation;
- `predictr()`: predicts response variable based on a `trainr()` model and input data;
- `trainr()`: primary function, trains a model based on training data and hyperparameters.

In addition to these functions there are also two internal functions `i2b()` and `b2i()`, these functions are used by `int2bin()` and `bin2int()` internally to change a single number from decimal to binary or visa versa.

The main `trainr()` function takes two arrays as inputs, the response variable `Y` and the input variable `X`, it returns a model that can be used with by the `predictr()` function together with a testing data input array `X`.

Internally, the functions make use of one or more sigmoid functions. In order to make the Sigmoid functions more generally available, these were moved to a separate package `sigmoid`.

The `sigmoid()` function is a wrapper, that defaults to the `logistic()` function, which maps the inputs to (0,1) using the logistic function, when using the default parameters, this is the standard logistic function.

## 2 Data

As mentioned above, an explicit element of Recurrent Neural Networks in the temporal aspect. As a result, both the input and the output can have up to three dimensions:

1. variables
2. observations

Table 2: sigmoid

```
# load library
library(sigmoid)

# list included functions
ls('package:sigmoid')

## [1] "Gompertz"                "SoftMax"
## [3] "inverse_Gompertz"        "logistic"
## [5] "logit"                   "sigmoid"
## [7] "sigmoid_output_to_derivative"
```

### 3. time periods

As a result of this, the functions take inputs of the type **array**, is a matrix is used as an input, the matrix is converted to an **array**. Conversely, if the output is a 2 dimensional, it will be simplified to a matrix.

## 2.1 Binary Addition

The vectors **X1** and **X2** are independent variables, the **Y** vector is the sum of **X1** and **X2** and acts as the response variable (for more info see `help('trainr')`).

Training data can be generated using **base** package's `sample()` function. For reproducibility, we also set the seed value of the pseudo-random number generator that **R** uses internally to 1. After generating **X1** and **X2**, I add the two pairwise and store the result in **Y**. Finally, I convert both the input variables and the response variable to binary representation using the `int2bin()` included with the package.

Table 3: Binary Numbers

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

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

# create training output
Y <- X1 + X2

# create 3d array: dim 1: samples; dim 2: time; dim 3: variables
X <- array( c(X1,X2), dim=c(dim(X1),2) )
```

Internally the `int2bin()` function converts these characters into binary format using the `intToBits()` function, the `bin2int()` function 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.

Table 4: Binary Representation

```
int2bin( X1[1] )

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    0    0    1    0    0    0    0    1
```

Lets check look at the first sum in decimal representation.

Table 5: Decimal Summation

```
X1[1]
## [1] 33

X2[1]
## [1] 89

X1[1] + X2[1]
## [1] 122

Y[1]
## [1] 122
```

and now in binary representation.

Table 6: Binary Summation

```
int2bin( X1[1] )
int2bin( X2[1] )
print('-----')
int2bin( Y[1] )

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    0    0    1    0    0    0    0    1
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    0    1    0    1    1    0    0    1
## [1] "-----"
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    0    1    1    1    1    0    1    0
```

As can be seen from the above output, the first values of X1 and X2, 33 and 89 respectively, are both in the range 0-127, which can be represented with only 7 bits. Yet the sum of the two - 122 - is almost outside of the range 0-127, . 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`).

We can now convert the entire vectors to binary matrices.

```
# convert to binaries of 8 bit (default)
X1 <- int2bin(X1)
X2 <- int2bin(X2)
Y  <- int2bin(Y)
```

## 2.2 Foreign Exchange Prediction

In this second example I train a RNN on more real-life data. I use the exchange rates of four major international currencies against the US Dollar.

Table 7: Foreign Exchange Data

```
library(quantmod) # for downloading FX data
start = '1998-12-14'
end   = '2001-09-01'

# download values
# output is automatically returned to
# the global environment (.GlobalEnv)
getFX('CHF/USD', from = start, to = end)
getFX('GBP/USD', from = start, to = end)
getFX('JPY/USD', from = start, to = end)
getFX('EUR/USD', from = start, to = end)
```

Input data should be on the domain  $[0,1]$ . Exchange rates are well suited for this since either A/B or B/A has to be in this domain. However, it is of course possible that within the time period studied, currency A, initially being worth less than currency B, becomes worth more. This is exactly what happened with the EUR/USD exchange rate. This means that neither EUR/USD nor USD/EUR is within the  $[0,1]$  domain for the entire period.

It is for this reason that sigmoid functions are used to map any real number to the domain  $[0,1]$ . The most function for this is the logistic function. At the end of the process, the outputs are mapped again to the original domain, using the inverse of the sigmoid function, in the case of the logistic, this is the logit function.

By specifying the `x0`.

## 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 `int2bin()` 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 into 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.

### 3.1 Binary Addition

The `trainr()` function will run until it has evaluated all rows in the matrices that it is fed and repeat this according to the number of epochs specified in the `numepochs` argument. 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).

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).

Table 8: `trainr()` Binary Addition

```
# train the network
m1 <- trainr(Y,
             X,
             hidden_dim      = 5,
             numepochs       = 10,
             learningrate    = 0.1,
             start_from_end = TRUE)

## Training epoch: 1 - Learning rate: 0.1
## Epoch error: 3.86763515257091
## Training epoch: 2 - Learning rate: 0.1
## Epoch error: 1.60548908339123
## Training epoch: 3 - Learning rate: 0.1
## Epoch error: 0.395638289496469
## Training epoch: 4 - Learning rate: 0.1
## Epoch error: 0.238016297019233
## Training epoch: 5 - Learning rate: 0.1
## Epoch error: 0.185323281721302
## Training epoch: 6 - Learning rate: 0.1
## Epoch error: 0.157134942538396
## Training epoch: 7 - Learning rate: 0.1
## Epoch error: 0.138868121568199
## Training epoch: 8 - Learning rate: 0.1
## Epoch error: 0.125779706036699
## Training epoch: 9 - Learning rate: 0.1
## Epoch error: 0.115801813496506
## Training epoch: 10 - Learning rate: 0.1
## Epoch error: 0.107867174922182
```

### 3.2 Foreign Exchange Prediction

The `trainr()` and `predictr()` functions internally make use of the `logistic()` function, specifically the standard logistic function, which takes the range  $(-\infty, \infty)$  and maps it to the range  $(0, 1)$ .



Table 9: Logistic Source Code

```
# print arguments to the function
args(logistic)

## function (x, k = 1, x0 = 0)
## NULL

# print source code of the logistic function
logistic

## function (x, k = 1, x0 = 0)
## 1/(1 + exp(-k * (x - x0)))
## <environment: namespace:sigmoid>
```

For instance:

Table 10: Logistic Examples

```
logistic(-137)
logistic(5.3)

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

The rough shape of the sigmoid function is shown below.

Figure 1: Standard Logistic Shape

```
library(ggplot2) # load plotting package

# sequence of -10 through 10
x = seq(-10, 10)

# plot sigmoid shape
qplot(x = x, y = logistic(x), geom='line')
```

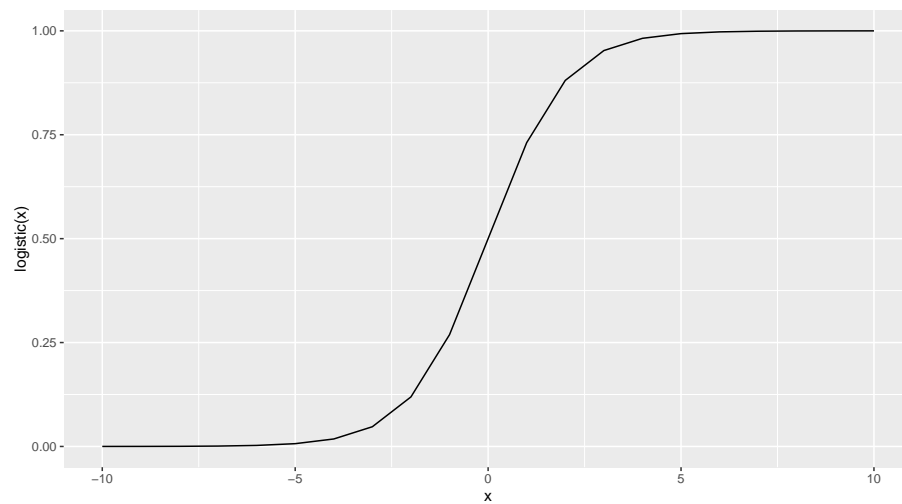


Figure 2: Logistic Mapping FX

```
# logistic map, write to new objects
chfUSD <- logistic(CHFUSD, k=sd(CHFUSD)^-1, x0=mean(CHFUSD) )
gbpUSD <- logistic(GBPUSD, k=sd(GBPUSD)^-1, x0=mean(GBPUSD) )
jpyUSD <- logistic(JPYUSD, k=sd(JPYUSD)^-1, x0=mean(JPYUSD) )
eurUSD <- logistic(EURUSD, k=sd(EURUSD)^-1, x0=mean(EURUSD) )
```

## 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.

## 4.1 Binary Addition Sums

Table 11: Binary Addition Test Data

```
# create test inputs
C1 <- int2bin( sample(0:127, 7000, replace=TRUE) )
C2 <- int2bin( sample(0:127, 7000, replace=TRUE) )

# stack matrices in array
C <- array( c(C1,C2), dim=c(dim(C1),2) )
```

Now predict using the `predictr()` function.

Table 12: `predictr()` Binary Addition

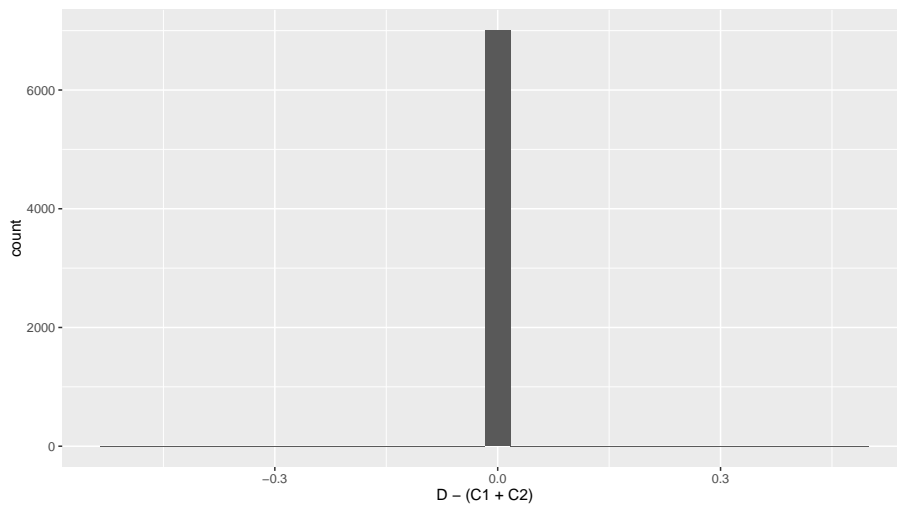
```
# predict
D <- predictr(model = m1, X = C )
```

We can now convert the predictions and the inputs back to decimals and plot them.

Figure 3: Binary Addition Sums

```
# convert back to decimal
C1 <- bin2int(C[,1])
C2 <- bin2int(C[,2])
D  <- bin2int(D)

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



As can be seen from the results, the difference is almost always 0.

## 4.2 Foreign Exchange Rate Predictions

Table 13: Foreign Exchange Rate Predictions

## 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 packages 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 Stuttgart 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 and C, 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

0\_Users\_quast\_rnn\_R\_trainr.R

```

1 #' @name trainr
2 #' @export
3 #' @importFrom stats runif
4 #' @importFrom sigmoid logistic sigmoid_output_to_derivative
5 #' @title Recurrent Neural Network
6 #' @description Trains a Recurrent Neural Network.
7 #' @param Y array of output values, dim 1: samples (must be equal
      to dim 1 of X), dim 2: time (must be equal to dim 2 of X), dim
      3: variables (could be 1 or more, if a matrix, will be coerce
      to array)
8 #' @param X array of input values, dim 1: samples, dim 2: time, dim
      3: variables (could be 1 or more, if a matrix, will be coerce
      to array)
9 #' @param learningrate learning rate to be applied for weight
      iteration
10 #' @param numepochs number of iteration, i.e. number of time the
      whole dataset is presented to the network
11 #' @param hidden_dim dimension of hidden layer
12 #' @param start_from_end should the sequence start from the end
13 #' @param learningrate_decay coefficient to apply to the learning
      rate at each weight iteration
14 #' @param momentum coefficient of the last weight iteration to keep
      for faster learning
15 #' @return a model to be used by the predictr function
16 #' @examples
17 #' # create training numbers
18 #' X1 = sample(0:127, 7000, replace=TRUE)
19 #' X2 = sample(0:127, 7000, replace=TRUE)
20 #'
21 #' # create training response numbers
22 #' Y <- X1 + X2
23 #'
24 #' # convert to binary
25 #' X1 <- int2bin(X1, length=8)
26 #' X2 <- int2bin(X2, length=8)
27 #' Y <- int2bin(Y, length=8)
28 #'
29 #' # create 3d array: dim 1: samples; dim 2: time; dim 3: variables
30 #' X <- array( c(X1,X2), dim=c(dim(X1),2) )
31 #'
32 #' # train the model
33 #' model <- trainr(Y=Y,
34 #'                 X=X,
35 #'                 learningrate = 0.1,
36 #'                 hidden_dim = 10,
37 #'                 start_from_end = TRUE )
38 #'
39
40 trainr <- function(Y, X, learningrate, learningrate_decay = 1,
      momentum = 0, hidden_dim, numepochs = 1, start_from_end=FALSE)
      {
41
42   # check the consistency

```

```

43   if(dim(X)[2] != dim(Y)[2]){
44       stop("The_time_dimension_of_X_is_different_from_the_time_
         dimension_of_Y.Only_sequences_to_sequences_is_supported")
45   }
46   if(dim(X)[1] != dim(Y)[1]){
47       stop("The_sample_dimension_of_X_is_different_from_the_sample_
         dimension_of_Y.")
48   }
49
50   # coerce to array if matrix
51   if(length(dim(X)) == 2){
52       X <- array(X,dim=c(dim(X),1))
53   }
54   if(length(dim(Y)) == 2){
55       Y <- array(Y,dim=c(dim(Y),1))
56   }
57
58   # extract the network dimensions
59   input_dim = dim(X)[3]
60   output_dim = dim(Y)[3]
61   binary_dim = dim(X)[2]
62
63   # initialize neural network weights
64   synapse_0 = matrix(stats::runif(n = input_dim*hidden_dim, min=-1,
        max=1), nrow=input_dim)
65   synapse_1 = matrix(stats::runif(n = hidden_dim*output_dim, min
        =-1, max=1), nrow=hidden_dim)
66   synapse_h = matrix(stats::runif(n = hidden_dim*hidden_dim, min
        =-1, max=1), nrow=hidden_dim)
67
68   # initialize the update
69   synapse_0_update = matrix(0, nrow = input_dim, ncol = hidden_dim)
70   synapse_1_update = matrix(0, nrow = hidden_dim, ncol = output_dim
        )
71   synapse_h_update = matrix(0, nrow = hidden_dim, ncol = hidden_dim
        )
72
73   # initialize the old update for the momentum
74   synapse_0_old_update = matrix(0, nrow = input_dim, ncol = hidden_
        dim)
75   synapse_1_old_update = matrix(0, nrow = hidden_dim, ncol = output
        _dim)
76   synapse_h_old_update = matrix(0, nrow = hidden_dim, ncol = hidden
        _dim)
77
78
79   # Storing layers states
80   store_output <- array(0,dim = dim(Y))
81   store_hidden <- array(0,dim = c(dim(Y)[1:2],hidden_dim))
82
83   # Storing errors, dim 1: samples, dim 2 is epochs, we could store
        also the time and variable dimension
84   error <- array(0,dim = c(dim(Y)[1],numepochs))
85
86   # training logic
87   for(epoch in seq(numepochs)){

```

```

88 message(paste0("Training_epoch:", epoch, " - Learning_rate:",
89               learningrate))
90
91 for (j in 1:dim(Y)[1]) {
92     # generate a simple addition problem (a + b = c)
93     a = array(X[j, , ], dim=c(dim(X)[2], input_dim))
94
95     # true answer
96     c = array(Y[j, , ], dim=c(dim(Y)[2], output_dim))
97
98     overallError = 0
99
100     layer_2_deltas = matrix(0, nrow=1, ncol = output_dim)
101     layer_1_values = matrix(0, nrow=1, ncol = hidden_dim)
102     # layer_1_values = rbind(layer_1_values, matrix(0, nrow=1,
103     #                                           ncol=hidden_dim))
104
105     # time index vector, needed because we predict in one
106     # direction but update the weight in an other
107     if(start_from_end == TRUE) {
108         pos_vec <- binary_dim:1
109         pos_vec_back <- 1:binary_dim
110     } else {
111         pos_vec <- 1:binary_dim
112         pos_vec_back <- binary_dim:1
113     }
114
115     # moving along the time
116     for (position in pos_vec) {
117
118         # generate input and output
119         x = a[position, ]
120         y = c[position, ]
121
122         # hidden layer (input + prev hidden)
123         layer_1 = sigmoid::logistic((x%%synapse_0) + (layer_1_
124           values[dim(layer_1_values)[1], ] %% synapse_h))
125
126         # output layer (new binary representation)
127         layer_2 = sigmoid::logistic(layer_1 %% synapse_1)
128
129         # did we miss?... if so, by how much?
130         layer_2_error = y - layer_2
131         layer_2_deltas = rbind(layer_2_deltas, layer_2_error *
132           sigmoid::sigmoid_output_to_derivative(layer_2))
133         overallError = overallError + sum(abs(layer_2_error))
134
135         # storing
136         store_output[j, position, ] = layer_2
137         store_hidden[j, position, ] = layer_1
138
139         # store hidden layer so we can print it out. Needed for
140         # error calculation and weight iteration
141         layer_1_values = rbind(layer_1_values, layer_1)
142     }
143 }

```



```

139 # store errors
140 error[j,epoch] <- overallError
141
142 future_layer_1_delta = matrix(0, nrow = 1, ncol = hidden_dim)
143
144 # Weight iteration,
145 for (position in 0:(binary_dim-1)) {
146
147     x = a[pos_vec_back[position+1],]
148     layer_1 = layer_1_values[dim(layer_1_values)[1]-
149         position,]
150     prev_layer_1 = layer_1_values[dim(layer_1_values)[1]-
151         position+1),]
152
153     # error at output layer
154     layer_2_delta = layer_2_deltas[dim(layer_2_deltas)[1]-
155         position,]
156     # error at hidden layer
157     layer_1_delta = (future_layer_1_delta %*% t(synapse_h) +
158         layer_2_delta %*% t(synapse_1)) *
159         sigmoid::sigmoid_output_to_derivative(layer_1)
160
161     # let's update all our weights so we can try again
162     synapse_1_update = synapse_1_update + matrix(layer_1) %*%
163         layer_2_delta
164     synapse_h_update = synapse_h_update + matrix(prev_layer_1)
165         %*% layer_1_delta
166     synapse_0_update = synapse_0_update + c(x) %*% layer_1_
167         delta # I had to change X as a vector as it is not a
168         matrix anymore, other option, define it as a matrix of
169         dim()=c(1,input_dim)
170
171     future_layer_1_delta = layer_1_delta
172 }
173
174 # Calculate the real update including learning rate and
175     momentum
176 synapse_0_update = synapse_0_update * learningrate + synapse_
177     0_old_update * momentum
178 synapse_1_update = synapse_1_update * learningrate + synapse_
179     1_old_update * momentum
180 synapse_h_update = synapse_h_update * learningrate + synapse_
181     h_old_update * momentum
182
183 # Applying the update
184 synapse_0 = synapse_0 + synapse_0_update
185 synapse_1 = synapse_1 + synapse_1_update
186 synapse_h = synapse_h + synapse_h_update
187
188 # Update the learning rate
189 learningrate <- learningrate * learningrate_decay
190
191 # Storing the old update for next momentum
192 synapse_0_old_update = synapse_0_update
193 synapse_1_old_update = synapse_1_update
194 synapse_h_old_update = synapse_h_update

```

```

183     # Initializing the update
184     synapse_0_update = synapse_0_update * 0
185     synapse_1_update = synapse_1_update * 0
186     synapse_h_update = synapse_h_update * 0
187 }
188 # update best guess if error is minimal
189 if (colMeans(error)[epoch] <= min(colMeans(error)[1:epoch])){
190     store_output_best <- store_output
191     store_hidden_best <- store_hidden
192 }
193 message(paste0("Epoch_error: ", colMeans(error)[epoch]))
194 }
195
196 # create utput object
197 output=list(synapse_0      = synapse_0,
198             synapse_1      = synapse_1,
199             synapse_h      = synapse_h,
200             error          = error,
201             store_output    = store_output,
202             store_hidden    = store_hidden,
203             store_hidden_best = store_hidden_best,
204             store_output_best = store_output_best,
205             start_from_end  = start_from_end)
206
207 attr(output, 'error') <- colMeans(error)
208
209
210 # return output
211 return(output)
212
213 }

```

## B Source code of predictr() function

```

1_Users_quast_rnn_R_predictr.R
1 #' @name predictr
2 #' @export
3 #' @importFrom stats runif
4 #' @importFrom sigmoid sigmoid
5 #' @title Recurrent Neural Network
6 #' @description predict the output of a RNN model
7 #' @param model output of the trainr function
8 #' @param X array of input values, dim 1: samples, dim 2: time, dim
9           3: variables (could be 1 or more, if a matrix, will be coerce
10              to array)
11 #' @param hidden should the function output the hidden units states
12 #' @param ... arguments to pass on to sigmoid function
13 #' @return array or matrix of predicted values
14 #' @examples
15 #' # create training numbers
16 #' X1 = sample(0:127, 7000, replace=TRUE)
17 #' X2 = sample(0:127, 7000, replace=TRUE)
18 #'
19 #' # create training response numbers

```

```

18 #' Y <- X1 + X2
19 #'
20 #' # convert to binary
21 #' X1 <- int2bin(X1)
22 #' X2 <- int2bin(X2)
23 #' Y <- int2bin(Y)
24 #'
25 #' # Create 3d array: dim 1: samples; dim 2: time; dim 3: variables
26
27 #' X <- array( c(X1,X2), dim=c(dim(X1),2) )
28 #'
29 #' # train the model
30 #' model <- trainr(Y=Y,
31 #'                  X=X,
32 #'                  learningrate = 0.1,
33 #'                  hidden_dim   = 10,
34 #'                  start_from_end = TRUE )
35 #'
36 #' # create test inputs
37 #' A1 = int2bin( sample(0:127, 7000, replace=TRUE) )
38 #' A2 = int2bin( sample(0:127, 7000, replace=TRUE) )
39 #'
40 #' # create 3d array: dim 1: samples; dim 2: time; dim 3: variables
41 #' A <- array( c(A1,A2), dim=c(dim(A1),2) )
42 #'
43 #' # predict
44 #' B <- predictr(model,
45 #'                A )
46 #'
47 #' # convert back to integers
48 #' A1 <- bin2int(A1)
49 #' A2 <- bin2int(A2)
50 #' B <- bin2int(B)
51 #'
52 #' # inspect the differences
53 #' table( B-(A1+A2) )
54 #'
55 #' # plot the difference
56 #' hist( B-(A1+A2) )
57
58 predictr <- function(model, X, hidden = FALSE, ...) {
59
60   # coerce to array if matrix
61   if(length(dim(X)) == 2){
62     X <- array(X,dim=c(dim(X),1))
63   }
64
65   # load neural network weights
66   synapse_0 = model$synapse_0
67   synapse_1 = model$synapse_1
68   synapse_h = model$synapse_h
69   start_from_end = model$start_from_end
70
71   # extract the network dimensions, only the binary dim
72   input_dim = dim(synapse_0)[1]
73   output_dim = dim(synapse_1)[2]

```

```

74 hidden_dim = dim(synapse_0)[2]
75 binary_dim = dim(X)[2]
76
77 # Storing layers states
78 store_output <- array(0,dim = c(dim(X)[1:2],output_dim))
79 store_hidden <- array(0,dim = c(dim(X)[1:2],hidden_dim))
80
81 for (j in 1:dim(X)[1]) {
82
83   # generate a simple addition problem (a + b = c)
84   a = array(X[j,,],dim=c(dim(X)[2],input_dim))
85
86
87   layer_1_values = matrix(0, nrow=1, ncol = hidden_dim)
88
89   # time index vector, needed because we predict in one direction
90   # but update the weight in an other
91   if(start_from_end == T){
92     pos_vec <- binary_dim:1
93     pos_vec_back <- 1:binary_dim
94   }else{
95     pos_vec <- 1:binary_dim
96     pos_vec_back <- binary_dim:1
97   }
98
99   # moving along the time
100   for (position in pos_vec) {
101
102     # generate input and output
103     x = a[position,]
104
105     # hidden layer (input ~+ prev hidden)
106     layer_1 = sigmoid::sigmoid((x**synapse_0) + (layer_1_values[
107       dim(layer_1_values)[1],] ** synapse_h), ...)
108
109     # output layer (new binary representation)
110     layer_2 = sigmoid::sigmoid(layer_1 ** synapse_1, ...)
111
112     # storing
113     store_output[j,position,] = layer_2
114     store_hidden[j,position,] = layer_1
115
116     # store hidden layer so we can print it out. Needed for error
117     # calculation and weight iteration
118     layer_1_values = rbind(layer_1_values, layer_1)
119
120   }
121 }
122
123 # return output vector
124 if(hidden == FALSE){
125   # convert to matrix if 2 dimensional
126   if(dim(store_output)[3]==1) {
127     store_output <- matrix(store_output,
128                           nrow = dim(store_output)[1],
129                           ncol = dim(store_output)[2]) }

```

```
128     # return output
129     return(store_output)
130 }else{
131     return(store_hidden)
132 }
133 }
```