`rnn`: a Recurrent Neural Network in R*

Bastiaan Quast†

11th 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.

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

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

Table 1: `rnn` Package

```r
# 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`.

Table 2: sigmoid

```r
# 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"
```

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
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
```

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

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

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

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

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

Table 8: `trainr()` arguments

```
args(trainr)

## function (Y, X, learningrate, learningrate_decay = 1, momentum = 0,
##     hidden_dim, numepochs = 1, start_from_end = FALSE)
## NULL
```

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 9: `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 (-Infinity, Infinity) and maps it to the range (0, 1).

Table 10: Logistic Source Code

```
# 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 11: 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

```r
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')
```

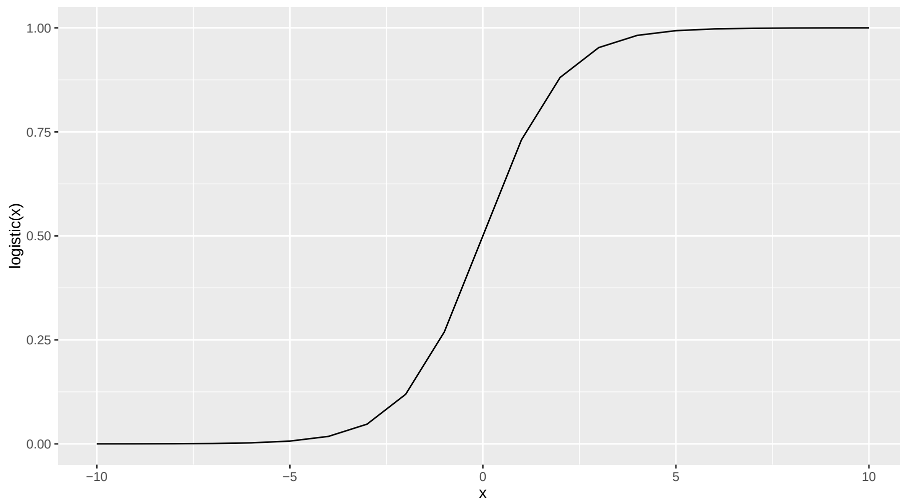

Figure 2: Logistic Mapping FX

```r
# 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 for new inputs.

4.1 Binary Addition Sums

Table 12: 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 13: `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
table( D - (C1+C2) )

##
##    0
## 7000
```

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

4.2 Foreign Exchange Rate Predictions

Table 14: 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 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 Stuttgart Neural Network Simulator library, which is written in C, and thereby makes available to Simple Recurrent Neural Networks 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

```r
#' @name trainr
#' @export
#' @importFrom stats runif
#' @importFrom sigmoid logistic sigmoid_output_to_derivative
#' @title Recurrent Neural Network
#' @description Trains a Recurrent Neural Network.
#' @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)
#' @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)
#' @param learningrate learning rate to be applied for weight iteration
#' @param numepochs number of iteration, i.e. number of time the whole dataset is presented to the network
#' @param hidden_dim dimension of hidden layer
#' @param start_from_end should the sequence start from the end
#' @param learningrate_decay coefficient to apply to the learning rate at each weight iteration
#' @param momentum coefficient of the last weight iteration to keep for faster learning
#' @return a model to be used by the predictr function
#' @examples
#' # create training numbers
#' X1 = sample(0:127, 7000, replace=TRUE)
#' X2 = sample(0:127, 7000, replace=TRUE)
#'
#' # create training response numbers
#' Y <- X1 + X2
#'
#' # convert to binary
#' X1 <- int2bin(X1, length=8)
#' X2 <- int2bin(X2, length=8)
#' Y  <- int2bin(Y,  length=8)
#'
#' # create 3d array: dim 1: samples; dim 2: time; dim 3: variables
#' X <- array( c(X1,X2), dim=c(dim(X1),2) )
#'
#' # train the model
#' model <- trainr(Y=Y,
#'                 X=X,
#'                 learningrate   =  0.1,
#'                 hidden_dim     = 10,
#'                 start_from_end = TRUE )
#'

trainr <- function(Y, X, learningrate, learningrate_decay = 1, momentum = 0, hidden_dim, numepochs = 1, start_from_end=FALSE) {

  # check the consistency
```

```
  if(dim(X)[2] != dim(Y)[2]){
    stop("The time dimension of X is different from the time dimension of Y. Only sequences to sequences is supported")
  }
  if(dim(X)[1] != dim(Y)[1]){
    stop("The sample dimension of X is different from the sample dimension of Y.")
  }

  # coerce to array if matrix
  if(length(dim(X)) == 2){
    X <- array(X,dim=c(dim(X),1))
  }
  if(length(dim(Y)) == 2){
    Y <- array(Y,dim=c(dim(Y),1))
  }

  # extract the network dimensions
  input_dim = dim(X)[3]
  output_dim = dim(Y)[3]
  binary_dim = dim(X)[2]

  # initialize neural network weights
  synapse_0 = matrix(stats::runif(n = input_dim*hidden_dim, min=-1, max=1), nrow=input_dim)
  synapse_1 = matrix(stats::runif(n = hidden_dim*output_dim, min=-1, max=1), nrow=hidden_dim)
  synapse_h = matrix(stats::runif(n = hidden_dim*hidden_dim, min=-1, max=1), nrow=hidden_dim)

  # initialize the update
  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)

  # initialize the old update for the momentum
  synapse_0_old_update = matrix(0, nrow = input_dim, ncol = hidden_dim)
  synapse_1_old_update = matrix(0, nrow = hidden_dim, ncol = output_dim)
  synapse_h_old_update = matrix(0, nrow = hidden_dim, ncol = hidden_dim)


  # Storing layers states
  store_output <- array(0,dim = dim(Y))
  store_hidden <- array(0,dim = c(dim(Y)[1:2],hidden_dim))

  # Storing errors, dim 1: samples, dim 2 is epochs, we could store also the time and variable dimension
  error <- array(0,dim = c(dim(Y)[1],numepochs))

  # training logic
  for(epoch in seq(numepochs)){
```

```
    message(paste0("Training epoch: ",epoch," - Learning rate: ",learningrate))
    for (j in 1:dim(Y)[1]) {

      # generate a simple addition problem (a + b = c)
      a = array(X[j,,],dim=c(dim(X)[2],input_dim))

      # true answer
      c = array(Y[j,,],dim=c(dim(Y)[2],output_dim))

      overallError = 0

      layer_2_deltas = matrix(0,nrow=1, ncol = output_dim)
      layer_1_values = matrix(0, nrow=1, ncol = hidden_dim)
      # layer_1_values = rbind(layer_1_values, matrix(0, nrow=1, ncol=hidden_dim))

      # time index vector, needed because we predict in one direction but update the weight in an other
      if(start_from_end == TRUE) {
        pos_vec      <- binary_dim:1
        pos_vec_back <- 1:binary_dim
      } else {
        pos_vec      <- 1:binary_dim
        pos_vec_back <- binary_dim:1
      }

      # moving along the time
      for (position in pos_vec) {

        # generate input and output
        x = a[position,]
        y = c[position,]

        # hidden layer (input ~+ prev_hidden)
        layer_1 = sigmoid::logistic((x%*%synapse_0) + (layer_1_values[dim(layer_1_values)[1],] %*% synapse_h))

        # output layer (new binary representation)
        layer_2 = sigmoid::logistic(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::sigmoid_output_to_derivative(layer_2))
        overallError = overallError + sum(abs(layer_2_error))

        # storing
        store_output[j,position,] = layer_2
        store_hidden[j,position,] = layer_1

        # store hidden layer so we can print it out. Needed for error calculation and weight iteration
        layer_1_values = rbind(layer_1_values, layer_1)

      }

```

```
    # store errors
    error[j,epoch] <- overallError

    future_layer_1_delta = matrix(0, nrow = 1, ncol = hidden_dim)

    # Weight iteration,
    for (position in 0:(binary_dim-1)) {

      x            = a[pos_vec_back[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(synapse_1)) *
        sigmoid::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 + c(x) %*% layer_1_delta # I had to change X as a vector as it is not a matrix anymore, other option, define it as a matrix of dim()=c(1,input_dim)

      future_layer_1_delta = layer_1_delta
    }

    # Calculate the real update including learning rate and momentum
    synapse_0_update = synapse_0_update * learningrate + synapse_0_old_update * momentum
    synapse_1_update = synapse_1_update * learningrate + synapse_1_old_update * momentum
    synapse_h_update = synapse_h_update * learningrate + synapse_h_old_update * momentum

    # Applying the update
    synapse_0 = synapse_0 + synapse_0_update
    synapse_1 = synapse_1 + synapse_1_update
    synapse_h = synapse_h + synapse_h_update

    # Update the learning rate
    learningrate <- learningrate * learningrate_decay

    # Storing the old update for next momentum
    synapse_0_old_update = synapse_0_update
    synapse_1_old_update = synapse_1_update
    synapse_h_old_update = synapse_h_update

```

```r
      # Initializing the update
      synapse_0_update = synapse_0_update * 0
      synapse_1_update = synapse_1_update * 0
      synapse_h_update = synapse_h_update * 0
    }
    # update best guess if error is minimal
    if(colMeans(error)[epoch] <= min(colMeans(error)[1:epoch])){
      store_output_best <- store_output
      store_hidden_best <- store_hidden
    }
    message(paste0("Epoch error: ",colMeans(error)[epoch]))
  }

  # create utput object
  output=list(synapse_0         = synapse_0,
              synapse_1         = synapse_1,
              synapse_h         = synapse_h,
              error             = error,
              store_output      = store_output,
              store_hidden      = store_hidden,
              store_hidden_best = store_hidden_best,
              store_output_best = store_output_best,
              start_from_end    = start_from_end)

  attr(output, 'error') <- colMeans(error)


  # return output
  return(output)

}
```

B Source code of `predictr()` function

1_Users_quast_rnn_R_predictr.R

```r
#' @name predictr
#' @export
#' @importFrom stats runif
#' @importFrom sigmoid sigmoid
#' @title Recurrent Neural Network
#' @description predict the output of a RNN model
#' @param model output of the trainr function
#' @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)
#' @param hidden should the function output the hidden units states
#' @param ... arguments to pass on to sigmoid function
#' @return array or matrix of predicted values
#' @examples
#' # create training numbers
#' X1 = sample(0:127, 7000, replace=TRUE)
#' X2 = sample(0:127, 7000, replace=TRUE)
#'
#' # create training response numbers
```

```
#' Y <- X1 + X2
#'
#' # convert to binary
#' X1 <- int2bin(X1)
#' X2 <- int2bin(X2)
#' Y  <- int2bin(Y)
#'
#' # Create 3d array: dim 1: samples; dim 2: time; dim 3: variables
  .
#' X <- array( c(X1,X2), dim=c(dim(X1),2) )
#'
#' # train the model
#' model <- trainr(Y=Y,
#'                 X=X,
#'                 learningrate   =  0.1,
#'                 hidden_dim     = 10,
#'                 start_from_end = TRUE )
#'
#' # create test inputs
#' A1 = int2bin( sample(0:127, 7000, replace=TRUE) )
#' A2 = int2bin( sample(0:127, 7000, replace=TRUE) )
#'
#' # create 3d array: dim 1: samples; dim 2: time; dim 3: variables
#' A <- array( c(A1,A2), dim=c(dim(A1),2) )
#'
#' # predict
#' B  <- predictr(model,
#'                A     )
#'
#' # convert back to integers
#' A1 <- bin2int(A1)
#' A2 <- bin2int(A2)
#' B  <- bin2int(B)
#'
#' # inspect the differences
#' table( B-(A1+A2) )
#'
#' # plot the difference
#' hist(  B-(A1+A2) )
#'

predictr <- function(model, X, hidden = FALSE, ...) {

  # coerce to array if matrix
  if(length(dim(X)) == 2){
    X <- array(X,dim=c(dim(X),1))
  }

  # load neural network weights
  synapse_0      = model$synapse_0
  synapse_1      = model$synapse_1
  synapse_h      = model$synapse_h
  start_from_end = model$start_from_end

  # extract the network dimensions, only the binary dim
  input_dim = dim(synapse_0)[1]
  output_dim = dim(synapse_1)[2]
```

```
  hidden_dim = dim(synapse_0)[2]
  binary_dim = dim(X)[2]

  # Storing layers states
  store_output <- array(0,dim = c(dim(X)[1:2],output_dim))
  store_hidden <- array(0,dim = c(dim(X)[1:2],hidden_dim))

  for (j in 1:dim(X)[1]) {

    # generate a simple addition problem (a + b = c)
    a = array(X[j,,],dim=c(dim(X)[2],input_dim))


    layer_1_values = matrix(0, nrow=1, ncol = hidden_dim)

    # time index vector, needed because we predict in one direction but update the weight in an other
    if(start_from_end == T){
      pos_vec <- binary_dim:1
      pos_vec_back <- 1:binary_dim
    }else{
      pos_vec <- 1:binary_dim
      pos_vec_back <- binary_dim:1
    }

    # moving along the time
    for (position in pos_vec) {

      # generate input and output
      x = a[position,]

      # hidden layer (input ~+ prev_hidden)
      layer_1 = sigmoid::sigmoid((x%*%synapse_0) + (layer_1_values[dim(layer_1_values)[1],] %*% synapse_h), ...)

      # output layer (new binary representation)
      layer_2 = sigmoid::sigmoid(layer_1 %*% synapse_1, ...)

      # storing
      store_output[j,position,] = layer_2
      store_hidden[j,position,] = layer_1

      # store hidden layer so we can print it out. Needed for error calculation and weight iteration
      layer_1_values = rbind(layer_1_values, layer_1)

    }
  }


  # return output vector
  if(hidden == FALSE){
    # convert to matrix if 2 dimensional
    if(dim(store_output)[3]==1) {
      store_output <- matrix(store_output,
                             nrow = dim(store_output)[1],
                             ncol = dim(store_output)[2])   }
```

```
        # return output
        return(store_output)
    }else{
        return(store_hidden)
    }
}
```