`rnn`: a Recurrent Neural Network in R*

Bastiaan Quast†

19th 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.

Table 1: `rnn` Package

```
# load the package
library(rnn)

# list functions
ls('package:rnn')

## [1] "bin2int"                      "int2bin"
## [3] "predictr"                     "sigmoid"
## [5] "sigmoid_output_to_derivative" "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;
- `sigmoid()`: converts any number to a probability between 0 and 1;
- `sigmoid_output_to_derivative()`: takes output of `sigmoid()` and returns the point derivative of that output;
- `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.

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 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 psuedo-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 2: Training Data

```r
# 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()` fucntion 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 3: Binary Representation

```r
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 4: Decimal Summation

```r
X1[1]

## [1] 33

X2[1]

## [1] 89

X1[1] + X2[1]

## [1] 122

Y[1]

## [1] 122
```

and now in binary representation.

Table 5: Binary Summation

```r
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, 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`).

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)
```

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 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 `trainr()` and `predictr()` functions internally make 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).

Table 6: Sigmoid Source Code

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

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

For instance:

Table 7: Sigmoid Examples

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

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

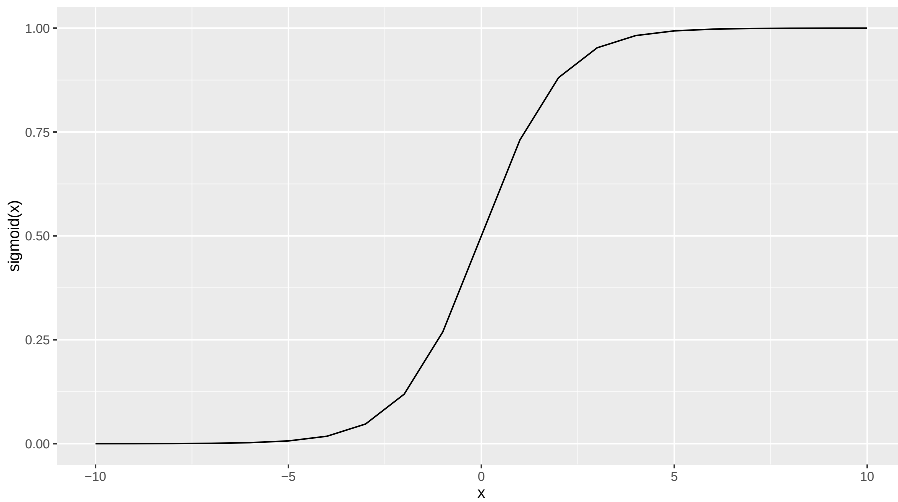
The shape of the sigmoid function is roughly shown below.

Figure 1: Sigmoid Shape

```r
library(ggplot2) # load plotting package

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

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



Additionally the `trainr()` and `predictr()` functions use the sigmoid_output_to_derivative() function.

Table 8: Sigmoid Derivative Source Code

```r
# 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).

Table 9: trainr() Output

```
## Error in 1:dim(Y)[1]:  argument of length 0
```

The `trainr()` function will run until it has evaluated all rows in the matrices 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`).

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 backpropagation (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:

Table 10: trainr() Output

```
# 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: 3.96355932085217"
## [1] "X1[ 1000 ]: 0 0 1 0 0 0 0 1   ( 33 )"
## [1] "X2[ 1000 ]: 0 1 1 0 0 1 0 0 + ( 100 )"
## [1] "----------------------------"
## [1] "Y[ 1000 ]:  1 0 0 0 0 1 0 1   ( 133 )"
## [1] "predict Y^: 1 1 0 0 1 1 0 1   ( 205 )"
## [1] "============================"
## [1] "Summation number: 2000"
## [1] "Error: 3.41811756435133"
## [1] "X1[ 2000 ]: 0 0 1 1 0 0 0 1   ( 49 )"
## [1] "X2[ 2000 ]: 0 0 1 1 0 0 1 1 + ( 51 )"
## [1] "----------------------------"
## [1] "Y[ 2000 ]:  0 1 1 0 0 1 0 0   ( 100 )"
## [1] "predict Y^: 0 0 0 0 0 0 1 0   ( 2 )"
## [1] "============================"
## [1] "Summation number: 3000"
## [1] "Error: 3.20534709314338"
## [1] "X1[ 3000 ]: 0 1 0 0 0 1 0 0   ( 68 )"
## [1] "X2[ 3000 ]: 0 1 0 1 0 0 1 1 + ( 83 )"
## [1] "----------------------------"
## [1] "Y[ 3000 ]:  1 0 0 1 0 1 1 1   ( 151 )"
## [1] "predict Y^: 1 0 0 1 1 1 1 1   ( 159 )"
## [1] "============================"
## [1] "Summation number: 4000"
## [1] "Error: 3.44514161374802"
## [1] "X1[ 4000 ]: 0 1 0 1 0 1 0 1   ( 85 )"
## [1] "X2[ 4000 ]: 0 0 1 1 0 1 1 1 + ( 55 )"
## [1] "----------------------------"
## [1] "Y[ 4000 ]:  1 0 0 0 1 1 0 0   ( 140 )"
## [1] "predict Y^: 1 1 1 0 1 0 0 0   ( 232 )"
## [1] "============================"
## [1] "Summation number: 5000"
## [1] "Error: 2.39543018899181"
## [1] "X1[ 5000 ]: 0 0 1 0 0 1 1 0   ( 38 )"
## [1] "X2[ 5000 ]: 0 0 1 0 1 0 1 1 + ( 43 )"
## [1] "----------------------------"
## [1] "Y[ 5000 ]:  0 1 0 1 0 0 0 1   ( 81 )"
## [1] "predict Y^: 0 1 0 1 1 0 0 1   ( 89 )"
## [1] "============================"
## [1] "Summation number: 6000"
## [1] "Error: 1.36041534642064"
## [1] "X1[ 6000 ]: 0 0 1 0 0 1 0 1   ( 37 )"
## [1] "X2[ 6000 ]: 0 1 1 0 1 1 0 1 + ( 109 )"
## [1] "----------------------------"
## [1] "Y[ 6000 ]:  1 0 0 1 0 0 1 0   ( 146 )"
## [1] "predict Y^: 1 0 0 1 0 0 1 0   ( 146 )"
## [1] "============================"
## [1] "Summation number: 7000"
## [1] "Error: 1.02655734929111"
## [1] "X1[ 7000 ]: 0 0 1 0 1 1 0 1   ( 45 )"
## [1] "X2[ 7000 ]: 0 1 0 1 1 1 0 0 + ( 92 )"
## [1] "----------------------------"
## [1] "Y[ 7000 ]:  1 0 0 0 1 0 0 1   ( 137 )"
## [1] "predict Y^: 1 0 0 0 1 0 0 1   ( 137 )"
## [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.

Table 11: Test Data

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

Now predict using the `predictr()` function.

Table 12: predictr()

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

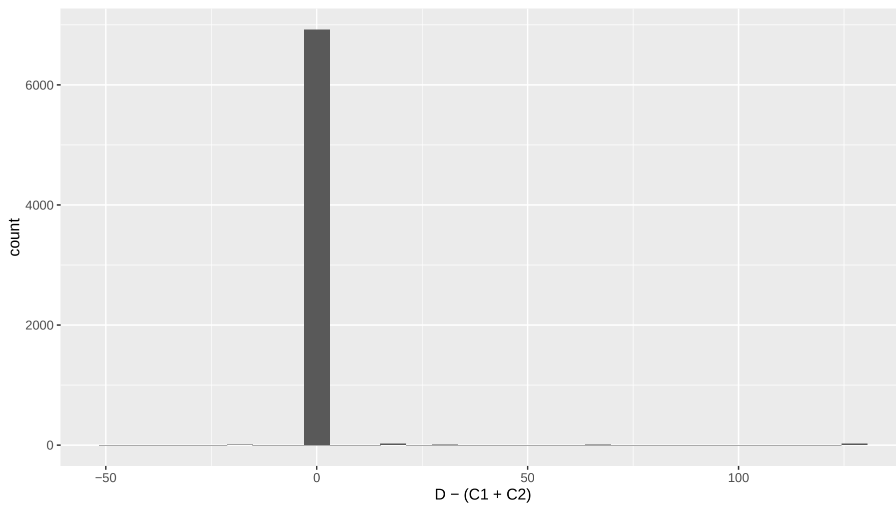
We can now convert the predictions and the inputs back to decimals and plot them.

Figure 2: Evalute Predictions

```
# convert back to decimal
C1 <- bin2int(C1)
C2 <- bin2int(C2)
D  <- bin2int(D)

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

## `stat_bin()` using `bins = 30`.  Pick better value with
`binwidth`.
```



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

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 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 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 `trainr()` function

0_Users_quast_rnn_R_trainr.R

```r
#' @name trainr
#' @export
#' @importFrom stats runif
#' @title Recurrent Neural Network
#' @description Trains a Recurrent Neural Network.
#' @param Y vector of output values
#' @param X1 vector of input values
#' @param X2 vector of input values
#' @param binary_dim dimension of binary representation
#' @param alpha size of alpha
#' @param input_dim dimension of input layer, i.e. how many numbers to sum
#' @param hidden_dim dimension of hidden layer
#' @param output_dim dimension of output layer
#' @param print should train progress be printed
#' @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)
#' X2 <- int2bin(X2)
#' Y  <- int2bin(Y)
#'
#' # train the model
#' trainr(Y,
#'        X1,
#'        X2,
#'        binary_dim =  8,
#'        alpha      =  0.1,
#'        input_dim  =  2,
#'        hidden_dim = 10,
#'        output_dim =  1,
#'        print = 'full'     )
#'


trainr <- function(Y, X1, X2, binary_dim, alpha, input_dim, hidden_dim, output_dim, print = c('none', 'minimal', '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_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)

  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:dim(Y)[1]) {

    if(print != 'none' && j %% 1000 == 0) {
      print(paste('Summation number:', j))
    }

    # generate a simple addition problem (a + b = c)
    a = X1[j,]
    b = X2[j,]

    # true answer
    c = Y[j,]

    # 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],] %*% synapse_h))

      # 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(layer_2))
      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(synapse_1)) *
      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

  # convert d to decimal
  out = b2i(as.vector(d))

  # print out progress
  if(print != 'none' && j %% 1000 == 0) {
    print(paste('Error:', overallError))
    print(paste('X1[', j, ']:', paste(a, collapse = ' '), ' ', '(', b2i(a), ')'))
    print(paste('X2[', j, ']:', paste(b, collapse = ' '), '+', '(', b2i(b), ')'))
```

```
      print('——————————————————————————————')
      print(paste('Y[', j, ']:␣', paste(c, collapse = '␣'), '␣', '(
          ', b2i(c), ')'))
      print(paste('predict␣Y^:',   paste(d, collapse = '␣'), '␣', '
          (', out, ')'))
      print('==============================')
    }
  }

  # output object with synapses
  return(list(synapse_0 = synapse_0, synapse_1 = synapse_1, synapse
      _h = synapse_h))

}
```

B Source code of `predictr()` function

1_Users_quast_rnn_R_predictr.R

```
#' @name predictr
#' @export
#' @importFrom stats runif
#' @title Recurrent Neural Network
#' @description Trains a Recurrent Neural Network.
#' @param model output of the trainr function
#' @param X1 vector of input values
#' @param X2 vector of input values
#' @param binary_dim dimension of binary representation
#' @param alpha size of alpha
#' @param input_dim dimension of input layer, i.e. how many numbers
     to sum
#' @param hidden_dim dimension of hidden layer
#' @param output_dim dimension of output layer
#' @param print should train progress be printed
#' @return vector 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)
#'
#' # train the model
#' m1 <- trainr(Y,
#'              X1,
#'              X2,
#'              binary_dim =  8,
#'              alpha      =  0.1,
#'              input_dim  =  2,
```

```
#'                hidden_dim = 10,
#'                output_dim =  1    )
#'
#' # create test inputs
#' A1 = int2bin( sample(0:127, 7000, replace=TRUE) )
#' A2 = int2bin( sample(0:127, 7000, replace=TRUE) )
#'
#' # predict
#' B  <- predictr(m1,
#'                  A1,
#'                  A2,
#'                  binary_dim =  8,
#'                  alpha      =  0.1,
#'                  input_dim  =  2,
#'                  hidden_dim = 10,
#'                  output_dim =  1    )
#'
#' # 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, X1, X2, binary_dim, alpha, input_dim,
    hidden_dim, output_dim, print = c('none', 'minimal', 'full')) {

  # check what largest possible number is
  largest_number = 2^binary_dim

  # create output vector
  Y <- matrix(nrow = dim(X1)[1], ncol = binary_dim)

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

  # training logic
  for (j in 1:dim(X1)[1]) {

    if(print != 'none' && j %% 1000 == 0) {
      print(paste('Summation number:', j))
    }

    # generate a simple addition problem (a + b = c)
    a = X1[j,]
    b = X2[j,]

    # 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])

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

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

      # 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^:', d[binary_dim - position]))
        print('=======')
      }
    }

    # output to decimal
    out = b2i(as.vector(d))

    # print out progress
    if(print != 'none' && j %% 1000 == 0) {
      print(paste('Error:', overallError))
      print(paste('X1[', j, ']:', paste(a, collapse = ' '), ' ', '(', b2i(a), ')'))
      print(paste('X2[', j, ']:', paste(b, collapse = ' '), '+', '(', b2i(b), ')'))
      print('------------------------------')
      print(paste('predict Y^:',   paste(d, collapse = ' '), ' ', '(', out, ')'))
      print('==============================')
    }

    # store value
    Y[j,] <- d
  }

  # return output vector
```

```
  return( Y )
}
```