`rnn`: Recurrent Neural Network architectures in native `R`

Bastiaan Quast*
International Telecommuniation Union

Dimitri Fichou
University of Giessen

June 22, 2018

Abstract

The R package `rnn` implements several Recurrent Neural Network (RNN) architectures in the R language. The native `R` implementations of these architectures allow scientists familiar with the `R` language, to develop an intuitive understanding of these architectures, something which is not possible with production frameworks, such as TensorFlow, PyTorch or CNTK.

1 About package `rnn` in `R`

The `rnn` package is available on CRAN at https://cran.r-project.org/package=rnn and can be installed using[1]:

```
install.packages('rnn')
```

After installation, the package can be loaded using:

```
library(rnn)
```

The following functions are exported by the package.

```
ls('package:rnn')

## [1] "bin2int"         "epoch_annealing"  "epoch_print"
## [4] "int2bin"         "loss_L1"          "predictr"
## [7] "run.finance_demo" "run.rnn_demo"    "trainr"
```

A list of all the functions - including non-exported ones - is shown below.

---

[1]The development version can be installed using `devtools::install_github('bquast/rnn')`

```
ls(getNamespace('rnn'), all.names=TRUE)
```

```
##  [1] ".__NAMESPACE__."      ".__S3MethodsTable__." ".packageName"
##  [4] "b2i"                  "backprop_gru"         "backprop_lstm"
##  [7] "backprop_r"           "backprop_rnn"         "bin2int"
## [10] "clean_lstm"           "clean_r"              "clean_rnn"
## [13] "epoch_annealing"      "epoch_print"          "i2b"
## [16] "init_gru"             "init_lstm"            "init_r"
## [19] "init_rnn"             "int2bin"              "loss_L1"
## [22] "predict_gru"          "predict_lstm"         "predict_rnn"
## [25] "predictr"             "run.finance_demo"     "run.rnn_demo"
## [28] "trainr"               "update_adagrad"       "update_r"
## [31] "update_sgd"
```

The **rnn** package has one dependency, the **sigmoid** package (Quast 2016), which is on CRAN at https://cran.r-project.org/package=sigmoid. The **sigmoid** package provides a collection of sigmoid functions such as the Rectified Linear Unit (`ReLU()`), `Gompertz()`, etc. Until version `0.8.0` of the **rnn** package, the sigmoid functions were included in the package, after which they were released as a separate package for more general use.

In addition to this, the **rnn** package includes a **Shiny** app demonstrating a Recurrent Neural Network analysis of a time series (Foreign Exchange rates). In order to run the app locally, the Shiny package needs to be installed.

2   trainr()

The workhorse of the **rnn** package is the **trainr()** function, it trains a model based on input and output data, given the specified hyperparameters.

The documentation of the **trainr()** function can be called up using:

```
help('trainr')
```

Recurrent Neural Networks (P. J. Werbos 1988) have the ability to learn bit-by-bit binary addition (including carrying over) with as little as 3 hidden nodes, whereas feed-forward neural networks would need many more.

First training data is generated, the training data is between 0-127, or an 8-bit binary.

```
set.seed(123) # for reproducible random numbers
X1 = sample(0:127, 50000, replace=TRUE)
X2 = sample(0:127, 50000, replace=TRUE)
```

The training data is used to generate the output data or labels.

```
Y <- X1 + X2
```

Following this, both the input data and the output data are converted into binary format, using the built-in `int2bin()` function.

```
X1 <- int2bin(X1, length=8)
X2 <- int2bin(X2, length=8)
Y  <- int2bin(Y,  length=8)
```

Finally, the two input variables are stored in a single 3 dimensional tensor. Where the first dimension contains observations, the second dimension time, and the third dimension variables.

```
X <- array( c(X1,X2), dim=c(dim(X1),2) )
```

The objects `X` and `Y` can now be fed to the `trainr()` function, which will output a trained model (stored here in the object called `m1`).

```
m1 <- trainr(Y=Y,
             X=X,
             learningrate   = 1,
             hidden_dim     = 6   )

## Trained epoch:  1 - Learning rate:  1
## Epoch error:   0.194449702806897
```

The forward and back propagation algorithms used are relatively straight-forward and can be printed.

```
rnn:::predict_rnn
rnn:::backprop_rnn
```

A simplied version of the rnn algorithm used is presented in the vignette **basic_rnn**, also available on CRAN at https://cran.r-project.org/package=rnn/vignettes/basic_rnn.html, it is also included in Appendix A.

```
vignette('basic_rnn')
```

3 predictr()

Using a trained model to make predictions is done using the `predictr()` function.

Observations `1` & `2` of the training data, the variable `1`, in binary format.

```
X[1:2,,1]
```

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

The same observations 1 & 2; variable 1, this time in decimal format.

```
bin2int( X[1:2,,1] )
```

```
## [1]  36 100
```

Observations 1 & 2 of the training data, the input variable 2, in binary format.

```
X[1:2,,2]
```

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

Observations 1 & 2; variable 2, in decimal format.

```
bin2int( X[1:2,,2] )
```

```
## [1] 27 56
```

Summing observation 1 of variable 1: 36, with observation 1 of variable 2: 119, gives 155.

Summing observation 2 of variable 1: 100, with observation 2 of variable 2: 75, gives 175.

Make predictions using the **predictr()** function.

```
round( predictr(model = m1,
                X     = X[1:2,,] ) )
```

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

Compared to the ground truth values.

```
Y[1:2,]
```

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

Or in decimal format.

```r
bin2int(round( predictr(model = m1,
                        X     = X[1:2,,] ) ) )
```

```
## [1]  63 156
```

Compared to the ground truth values.

```r
bin2int( Y[1:2,] )
```

```
## [1]  63 156
```

4   Architectures

In addition to fully-connected Recurrent Neural Networks, `rnn` also supports Long Short-Term Memory (LSTM) Recurrent Neural Networks (Hochreiter and Schmidhuber 1997)

```r
trainr(Y, X, network_type="lstm")
```

as well as Gated Recurrent Unit (Cho et al. 2014) architecture.

```r
trainr(Y, X, network_type="gru")
```

5   Miscellaneous Functions

In addition to the main user APIs `trainr()` and `predictr()`, the underlying functions for the above mentioned neural network architectures, the `rnn` package includes several other functions. Including the `int2bin()` and `bin2int()` which were used above, these funtions are intended to ease conversion from decimal to binary notation and visa versa, especially for didacting purposes.

The `run.rnn_demo()` and `run.finance_demo()` each launch a Shiny app that demonstate usage of `rnn` functionality, the Shiny package needs to be installed in order to run these apps locally[2]. The two demos are also available online at http://shiny.qua.st/rnn and http://shiny.qua.st/finance respectively.

References

Cho, Kyunghyun et al. (2014). "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078*.

---

[2]The Shiny package can be installed using: `install.packages('shiny')`

Hebb, Donald Olding (2005). *The organization of behavior: A neuropsychological theory*. Psychology Press.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long short-term memory". In: *Neural computation* 9.8, pp. 1735–1780.

Linnainmaa, Seppo (1970). "The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors". In: *Master's Thesis (in Finnish), Univ. Helsinki*, pp. 6–7.

McCulloch, Warren S and Walter Pitts (1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133.

Minsky, Marvin (1952). "A neural-analogue calculator based upon a probability model of reinforcement". In: *Harvard University Psychological Laboratories, Cambridge, Massachusetts*.

Quast, Bastiaan (2016). *sigmoid: Sigmoid Functions for Machine Learning*. R package version 0.3.0. URL: https://cran.r-project.org/package=sigmoid.

Rosenblatt, Frank (1958). "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6, p. 386.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1986). "Learning representations by back-propagating errors". In: *nature* 323.6088, p. 533.

Werbos, Paul (1974). "Beyond regression: new fools for prediction and analysis in the behavioral sciences". In: *PhD thesis, Harvard University*.

Werbos, Paul J (1988). "Generalization of backpropagation with application to a recurrent gas market model". In: *Neural networks* 1.4, pp. 339–356.

Widrow, Bernard et al. (1960). *Adaptive" adaline" Neuron Using Chemical" memistors."*.

A   Simplied RNN code

Decimal to binary conversion.

```r
i2b <- function(integer, length=8)
  as.numeric(intToBits(integer))[1:length]

# apply to entire vectors
int2bin <- function(integer, length=8)
  t(sapply(integer, i2b, length=length))
```

Training data generation.

```r
# set training data length
training_data_size = 20000

# create sample inputs
```

```r
X1 = sample(0:127, training_data_size, replace=TRUE)
X2 = sample(0:127, training_data_size, replace=TRUE)

# create sample output
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) )
```

Sigmoid and derivative functions.

```r
sigmoid <- function(x)
            1 / ( 1+exp(-x) )

sig_to_der <- function(x)
              x*(1-x)
```

Set the hyperparameters.

```r
binary_dim = 8
alpha      = 0.5
input_dim  = 2
hidden_dim = 6
output_dim = 1
```

Initialise the weights.

```r
# initialize weights randomly between -1 and 1, with mean 0
weights_0 = matrix(runif(n = input_dim *hidden_dim, min=-1, max=1),
                   nrow=input_dim,
                   ncol=hidden_dim )
weights_h = matrix(runif(n = hidden_dim*hidden_dim, min=-1, max=1),
                   nrow=hidden_dim,
                   ncol=hidden_dim )
weights_1 = matrix(runif(n = hidden_dim*output_dim, min=-1, max=1),
                   nrow=hidden_dim,
                   ncol=output_dim )

# create matrices to store updates, to be used in backpropagation
weights_0_update = matrix(0, nrow = input_dim,  ncol = hidden_dim)
```

```r
weights_h_update = matrix(0, nrow = hidden_dim, ncol = hidden_dim)
weights_1_update = matrix(0, nrow = hidden_dim, ncol = output_dim)
```

Train the model.

```r
# training logic
for (j in 1:training_data_size) {
    # select data
    a = X1[j,]
    b = X2[j,]

    # select 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)

    # moving along the positions in the binary encoding
    for (position in 1:binary_dim) {
        # generate input and output
        X = cbind( a[position], b[position] )
        y = c[position]

        # hidden layer
        layer_1 = sigmoid( (X%*%weights_0) +
                    (layer_1_values[dim(layer_1_values)[1],]
                                            %*% weights_h) )

        # output layer
        layer_2 = sigmoid(layer_1 %*% weights_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 * sig_to_der(layer_2))
        overallError = overallError + abs(layer_2_error)

        # decode estimate so we can print it out
        d[position] = round(layer_2)
```

```r
        # store hidden layer
        layer_1_values = rbind(layer_1_values, layer_1)
    }

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

    for (position in binary_dim:1) {
        X = cbind(a[position], b[position])
        layer_1 = layer_1_values[dim(layer_1_values)[1]
                                    - (binary_dim-position),]
        prev_layer_1 = layer_1_values[dim(layer_1_values)[1]
                                    - ( (binary_dim-position)+1 ),]

        # error at output layer
        layer_2_delta = layer_2_deltas[dim(layer_2_deltas)[1] -
                                    (binary_dim-position),]
        # error at hidden layer
        layer_1_delta = (future_layer_1_delta %*% t(weights_h) +
            layer_2_delta %*% t(weights_1)) * sig_to_der(layer_1)

        # let's update all our weights so we can try again
        weights_1_update = weights_1_update + matrix(layer_1) %*%
                                            layer_2_delta
        weights_h_update = weights_h_update + matrix(prev_layer_1) %*%
                                            layer_1_delta
        weights_0_update = weights_0_update + t(X) %*% layer_1_delta

        future_layer_1_delta = layer_1_delta
    }

    weights_0 = weights_0 + ( weights_0_update * alpha )
    weights_1 = weights_1 + ( weights_1_update * alpha )
    weights_h = weights_h + ( weights_h_update * alpha )

    weights_0_update = weights_0_update * 0
    weights_1_update = weights_1_update * 0
    weights_h_update = weights_h_update * 0

    if(j%%(training_data_size/5) == 0)
        print(paste("Error:", overallError))

}
## [1] "Error: 2.70103303147726"
## [1] "Error: 0.273505476217851"
## [1] "Error: 0.158342968262358"
```

```
## [1] "Error: 0.139704536271053"
## [1] "Error: 0.177159616857211"
```