Student Application for R in GSoC-2016

Yun Yan

March 18, 2016

1 PROJECT INFO

Project title: **Deep learning with MXNet**

Project short title: Implement high-level interfaces (APIs) of Recurrent Neural Network (RNN) of MXNet to achieve user-friendly network construction for R community.

URL of project idea page: https://github.com/rstats-gsoc/gsoc2016/wiki/Deep-learning-with-mxnet

2 BIO OF STUDENT

- Yun is a graduate student majoring in Computer Science at New York University with emphasis on data science. It is not surprising that he is not only familiar with regular conventions (e.g. Git) and programming languages (e.g. C++) of software development, but also has solid knowledge of theories related to neural network and other machine learning applications. As a proof-of-concept to implement high-level API of LSTM which is one of MXNet's GitHub issues, he had already built a many-to-many RNN model (see Gist) from scratch which managed to learn 8-bit binary addition calculation. Model was implemented in Rcpp which fits the language requirement of this project. Therefore the CS background supports Yun to read the source code of MXNet, communicate with mentors, identify important but absent features, and ultimately finish implementation.
- Yun is an experienced R user who has been participating in developing R package. For example, ChIPseeker, a Bioconductor package for bioinformatics analysis; honfleuR, an extension supporting object-oriented programming in R S4 methods for an existed R package to analyze single-cell sequencing data. Therefore his strong interest and 4-year experience in R language make him a self-motivated and competent participant to help make a powerful deep learning package dedicated to R language community.
- Yun has rich research experience in computational biology with a Master degree in Molecular Biology. He realized deep learning is becoming a promising strategy to analyze large scale of data produced by biomedical studies. Therefore Yun has solid expert-domain knowledge and strong interest to apply deep learning to computational biology field by writing case studies or examples for MXNet package. Therefore Yun's participation can bring to new impacts – deep learning has a soft-landing on life sciences researches thanks to MXNet's R package and likewise MXNet becomes more appealing to bioinformaticians and broaden user spectrum.

3 CONTACT INFORMATION

Student name: Yun Yan

Student postal address: 334 E 25 St, Apt 213, New York, NY, 10010

Telephone(s): 917-756-3868

Email(s): yy1533@nyu.edu (primary account); youryanyun@gmail.com (acount for sharing draft).

3.1 Student affiliation

Institution: New York University

Program: Master program, Computer Science, Tandon School of Engineering

Stage of completion: 2015.09 - 2017.06

Contact to verify: Office of Global Services (OGS), 5 MetroTech Center, Room 259, Brooklyn, NY 11201. Tel: (646) 997-3805

3.2 Schedule Conflicts:

Off-keyboards on Sundays, otherwise there is no time schedule conflicts. I am dedicated to GSoC this summer.

4 MENTORS

4.1 Mentor names

Qiang Kou (qkou@umail.iu.edu), Yuan Tang (terrytangyuan@gmail.com)

4.2 Contact with Mentors

Date	Event	Media
17-Feb-2016	MXNet initiated idea page on rstats-GSoC	NA
24-Feb-2016	Finished a proof-of-concept of RNN implemented in Rcpp and contact mentors	Github
2-Mar-2016	Submitted "Pull Request" as qualification of mentor test	Github
2-Mar-2016	Contact mentors to express my interest in MXNet's GSoC project	Email
3-Mar-2016	My PR being merged	Github
8-Mar-2016 ~ 12-Mar-2016	Drafts of student application form	Email&Github
13-Mar-2016	Finish draft to be reviewed by mentor	GitHub

GitHub repository where discussions with mentors and proposal LaTeX/markdown source codes are saved: https://github.com/Puriney/mxnet-gsoc-yunyan-application

5 WHY THIS PROPOSAL

Enhancing R package of MXNet is going to provide R community a swift deep learning framework. Supports for either advanced structure or accepted performance are not fulfilled by existed R package for deep learning, for example, nnet and deepnet. On the contrary, in Kaggle Data Science Competition, MXNet is getting popular among kagglers thanks to its ability (i.e. basic APIs of R package) to allow R users not only implement deep learning model without getting hands too wet in programming codes, but also train networks in affordable time via supporting GPU-accelerated. Therefore, the work proposed here is going to push MXNet a further step to better serve for R community.

High-level APIs for advanced structures are absent. Admittedly existed basic functions of MXNet's R package are so modularized, self-explaining, well-documented that could be used in combinations to design advanced networks, for example, LSTM (long-short term memory network) is built by MXNet's python APIs and shown in official tutorial. However, what if MXNet supports high-level APIs for LSTM, GRU, bidirectional RNN and other types of RNN dedicated to R community, R users could focus on data analysis and problem solving, rather than figuring out how to build advanced networks first. And increasing needs (see issue) for high-level APIs of RNN are reported.

6 EXPECTED APIs AND IMPACTS

I am planing to implement high-level functions for the following types of RNN (recurrent neural network) in light of its popularity in data science and absence in MXNet. There exists `mx.symbol.Convolution` for CNN already.

1. vanilla RNN
2. LSTM
3. GRU
4. bi-directional RNN
5. multi-layer RNN

The expected results should enable R users swiftly design a neural network. For example, a model with vanilla RNN as hidden layer is expected to be constructed in following 3 steps rather than hundreds lines of R codes:

```
data <- mx.symbol.Variable("data")
lay1 <- mx.symbol.vanillaRNN(data, num_hidden = 16)
vrnn <- mx.symbol.Softmax(lay1)
```

The expected impact is that R community could not only keep enjoying the MXNet's flexibility as it wisely incorporates symbolic configuration and imperative programming together, but also have an end-to-end deep learning framework without being distracted by thinking of coding details.

7 CODING PLANS AND METHODS

The project requires student with programming skills on R/Cpp/Rcpp and solid knowledge of deep learning.

7.1 Proof-of-concept

As a proof of concept, I had already implemented RNN to learn 8-bit binary calculus using Rcpp starting from scratch (See Gist), e.g. learning $01111001 + 00010101 = 10001110$ for $121 + 21 = 142$. Because the each bit is equivalent to timestamp, shown as following figure, my proof-of-concept implementation was indeed equivalent to and did manage to fulfill a synced many-to-many RNN construction (See Fig-1) .

Fortunately, model construction is achieved by symbolic configuration in MXNet. Thus compared with imperative programming (shown as proof-of-concept above), I have more convenient and feasible approach to implement advanced models. Things becomes even more easier given that I have knowledge about the model's topological structure.

The topological structures of variations of RNN are listed below, and the model listed later depend on the ones stated earlier. Thus my coding plan is step-by-step, model-by-model, bottom-up.

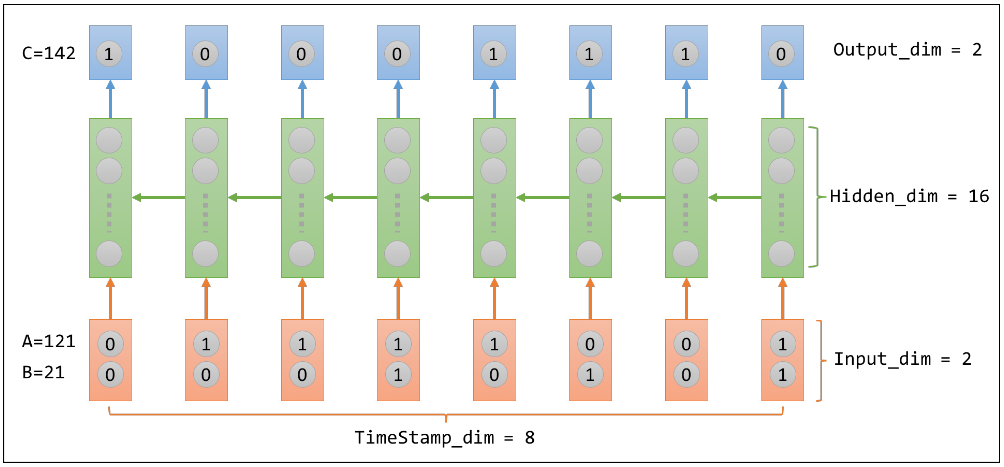

Figure 1: Diagram showing proof-of-concept where RNN learns 8 bit binary addition. The green arrows indicating the direction of forward propagation within hidden layer are marked as right-to-left for better illustration of binary addition. R/Rcpp source codes are shared on Gist or shown as in Appendices part.

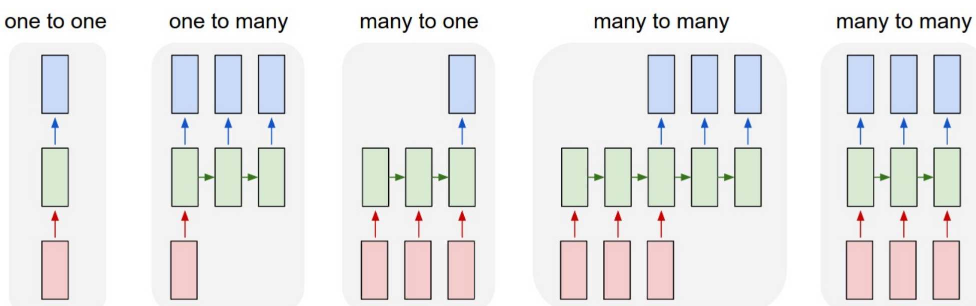

Figure 2: Diagram showing basic types of RNN. Red, green, blue blocks indicate input layer, hidden layer, output layer. And input/output layer has optional time stamp settings. Source: The Unreasonable Effectiveness of Recurrent Neural Networks.

7.2 Four types of vanilla RNN

The expected snippet of calling vanilla RNN is below:

```
mx.symbol.vanillaRNN(symbol, name, hidden_dim, type = c('1toN', 'Nto1', 'NtoN', 'syncNtoN'), ...)
```

My proof-of-concept is imperative programming while the network in built by symbolic configuration in MXNet. Therefore I plan to implement high-level API for synced many-to-many RNN at first.

Once imperative-to-declarative translation were finished, solutions for composing graph of the rest three categories of RNN are expected to be straightforward, in light of the fact that these three are derivations of synced many-to-many RNN, shown as Fig-3 .

7.3 LSTM

LSTM is specific type of RNN, while it is independently listed given its popularity.

It has 3 types of gate layers to control the data flow within layers: forgot, input, and output gate layers.

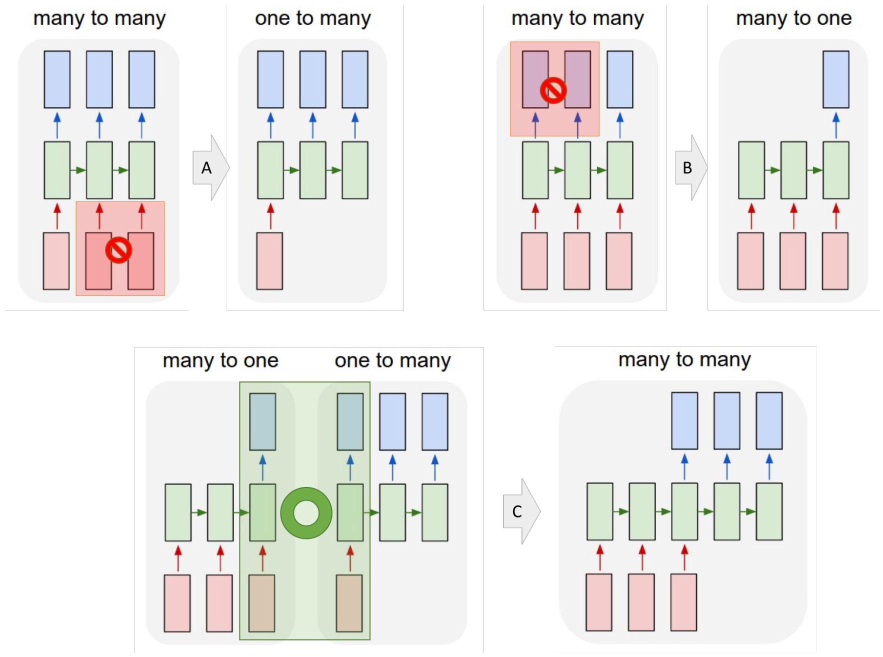

Figure 3: Operation A: suppress part of input; Operation B: suppress part of output; Operation C: combine/group the two layers highlighted in green rectangular. Figures are modified from original figure

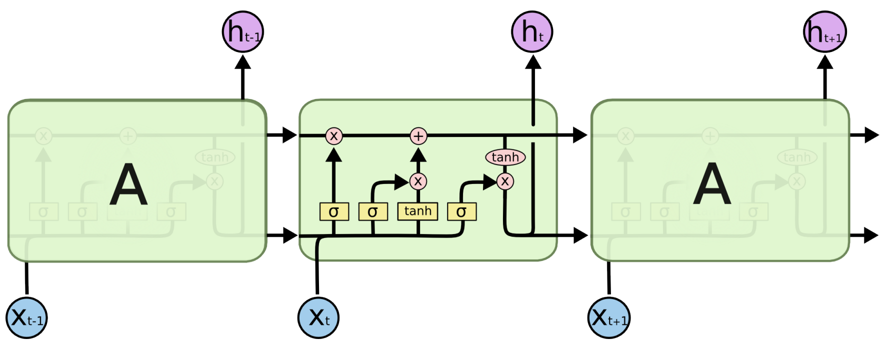

Figure 4: The underlying structure of LSTM. Source: Understanding LSTM Networks

The developer of MXNet's Julia package once posted a step-by-step tutorial for constructing LSTM, in addition to established MXNet's python demos for LSTM. Therefore I am expected to follow the logics to implement symbol operation for LSTM. Implementation for LSTM API is expected to be easiest task throughout this project.

7.4 GRU

GRU is derivation of LSTM. It combines input gate and forget gate combined as update gate, and comes up with other minor modifications, shown as Fig-5.

Following the LSTM implementation, GRU is straightforward.

7.5 Bidirectional RNN

Within this computation graph, forward propagation and backward propagation run through their own layer.

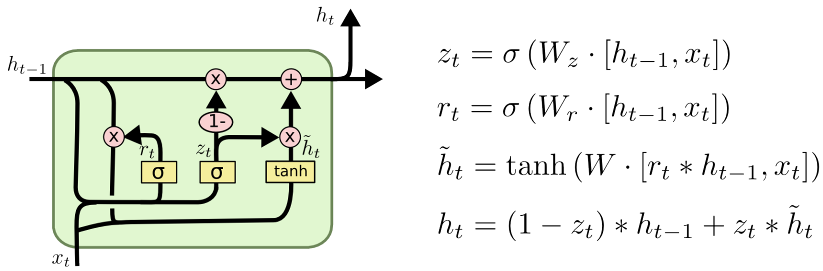

Figure 5: Diagram illustrating underlying structure of GRU. Source: Understanding LSTM Networks

Ref: Hybrid Speech Recognition with. Deep Bidirectional LSTM

7.6 Multi-layer RNN

RNNs are stacked. Self-explaining (See Fig-6 .

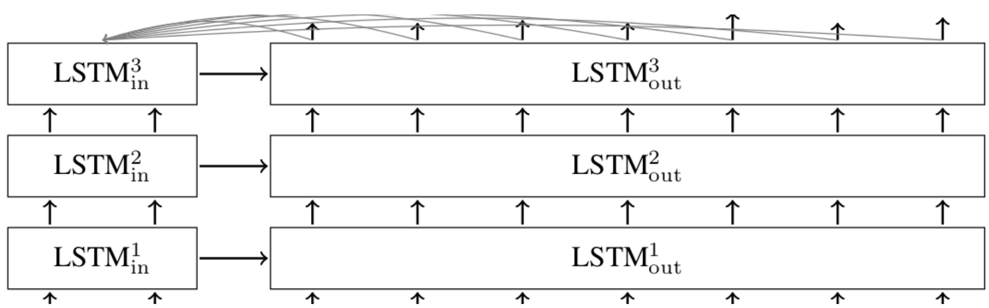

Figure 6: Stacked LSTM for attention-based seq2seq model. See source post of TensorFlow

7.7 Application in computational biology

I would like to reproduce the CNN model published on Nature magazine in order to draw attention of R users in bioinformatics, computational biology field. The results of this possible extra work can also be checked whether valid by mentors as Qiang Kou had experience in bioinformatics.

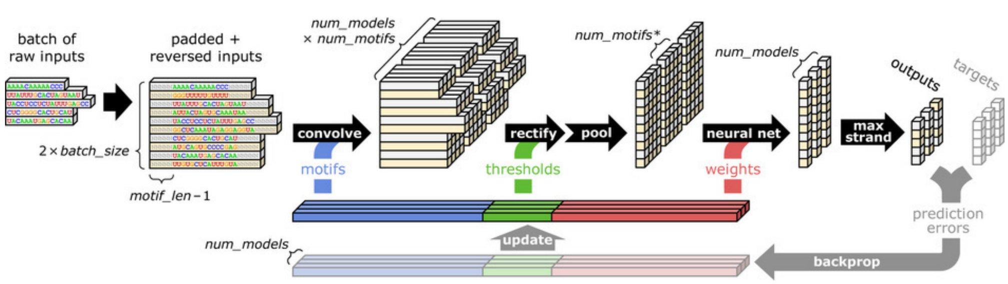

Figure 7: Source: Predicting the sequence specificities of DNA- and RNA-binding proteins by deep learning

8 PERCEIVED OBSTACLES

The critical turn-overs are:

1. How to precisely adapt my knowledge and existed imperative implementation (as the proof-of-concept mentioned before) to MXNet's high-level APIs, so that R users could enjoy symbolic configuration.
2. How to make APIs support performance as much as possible.

For first part, I could wrap up all basic interfaces to write functions with parameters to support user-defined network structures, which is the actual implementation strategy of `mx.mlp()` layer function for multilayer perceptron.

If necessary, I would make modifications for *R/src/symbol.cc* etc to meet development needs. For example, MXNet's develop modified *symbol.cc* to solve issue reported here which asked for channel slicing to build LSTM.

In addition, basic interfaces contain parameters related to GPU, wrapper function could take care of performance.

Therefore wrapper-function is going to be qualified and feasible strategy.

For second part which is native implementation, fortunately MXNet team maintains a well-documented guide for new developers, in particular the following posts stated the rules and conventions I need to follow:

1. How to Create New Operations (Layers)
2. Operators in MXNet

For example, the implementation of CNN layer is via three source files located at directory *src/operator/* : `convolution-inl.h`, `convolution.cc`, `convolution.cu`. In the similar way, I could implement the new operators for RNN, if necessary. Furthermore, MXNet's R package has already set up the fundamental interfaces thus my work will not start from scratch.

Putting things together, I could have feasible approaches to implement high-level APIs for RNNs and handle perceived problems in affordable time.

9 TIMELINE

9.1 Pre-coding period

Start	End	Topic
25-Apr-2016	1-May-2016	Read documents. Get familiar with **details** of code structure for neural network symbol.
2-May-2016	8-May-2016	Read documents. Get familiar with **details** of code structure of tensor computation in mxnet so that I could better adapt my RNN codes for appropriate forward/backward propagation.
9-May-2016	16-May-2016	NYU Final exam season.
17-May-2016	22-May-2016	Contact with mentors to get started. Set up formal communication tools e.g. Gitter.

9.2 Coding Period

The following 4 jobs must be done when working on every API.

1. Implementation of high-level API;
2. **Testing** (see details of codes tests as sections below);

3. **Write demos and case studies**;
4. Submit codes and discuss with mentors to see whether they are ready to be merged into main stream.

Start	End	APIs	Demo	Days
23-May-2016	5-Jun-2016	Synced Many2Many	Binary Addition	12
6-Jun-2016	16-Jun-2016	One2Many	Image Classification	10
17-Jun-2016	26-Jun-2016	Many2One	Char RNN	8
27-Jun-2016	5-Jul-2016	Many2Many	Translation English to French	8
6-Jul-2016	14-Jul-2016	Bidirectional	Translation English to French	8
15-Jul-2016	24-Jul-2016	Stacked	Translation English to French	8
25-Jul-2016	10-Aug-2016	LSTM and GRU	Char RNN	15

Start	End	Application	Reference	Days
11-Aug-2016	21-Aug-2016	Case Study of computational biology	Nature Paper	9

(*Days* column excludes Sundays)

In sum, during working period I am glad to actively communicate with mentors to input more qualified codes into MXNet's R package in addition to works scheduled here, e.g. more demos and case studies for users, other APIs not listed here, etc.

9.3 Post-coding period

22-Aug-2016: Wrapping up the entire project. Because project is finishing API one-by-one, documents and tests are expected to qualified and final wrapping up will not take too long.

From 23-Aug-2016 to 29-Aug-2016: Final Evaluation.

10 MANAGEMENT OF CODING PROJECT

10.1 Where are codes deployed

Fork of MXNet's repository: https://github.com/Puriney/mxnet.

10.2 How to test codes

10.2.1 Travis Test

My codes can directly use the Travis tests currently used by MXNet's main repository. Once passing, they are always ready to be merged into main stream.

10.2.2 Fast Test

As it is R package, I could use the following snippet to test the R package.

```
R CMD check --no-examples --no-manual --no-vignettes --no-build-vignettes mxnet_*.tar.gz
```

In addition, thanks to roxygen2, possible conflicts will be reported when it generates documents for functions.

10.2.3 Test against real data

Writing case studies and demos for functions are good conventions of MXNet and I am expected to follow. In MXNet repository, there existed data of Penn Treebank Project for RNN, thus I could run test by applying these data on my codes.

10.3 Expected Commits Frequency

Commits will be pushed in every 2 days. Commits within a working period (6 days) are squashed as one commit to make history clean and friendly to be ready to be merged into MXNet's main repository.

Being absent for 10 days suggests I must come across with problems.

11 TEST

I fixed an issue and the pull request was afterwards **merged** (See: https://github.com/dmlc/mxnet/pull/1554) into MXNet main repository, thus I was marked as potential student for this project.

It supports following features:

- Xavier strategy to initialize weights
- Clipping gradient (i.e. fixing calculated gradient within a range)
- Scheduling mini-changes for learning rate value along training process.

Passing the tests suggest that I am familiar with MXNet's codes structure and ready to participate in R language project of MXNet in GSoC-2016 with matched skills.

Appendices

```
//
// Yun Yan
//
// [[Rcpp::plugins(cpp11)]]
#include <bitset>
#include <unordered_set>
#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]

// #include <RcppEigen.h>
//// [[Rcpp::depends(RcppEigen)]]

using namespace Rcpp;
// using namespace Eigen;
using namespace arma;
```

```
// Using Rcpp to reproduce "RNN in python"

//' Sigmoid function
//'
//' @param x A numeric vector.
//' @export
// [[Rcpp::export]]
NumericVector sigmoid(NumericVector x) {
  NumericVector ret = 1 / (1 + exp(-1 * x));
  return ret;
}

//' First derivative of sigmoid function
//'
//' @param x A numeric vector.
//' @export
// [[Rcpp::export]]
NumericVector sigmoid_deriv(NumericVector x) {
  return x * (1 - x);
}

// [[Rcpp::export]]
NumericMatrix dotprodmm(NumericMatrix a, NumericMatrix b) {
  // dotprod = matrix %*% matrix
  mat aa = as<mat>(a);
  mat bb = as<mat>(b);
  mat cc = aa * bb;
  return(wrap(cc));
}

// [[Rcpp::export]]
NumericMatrix dotprodvm(NumericVector a, NumericMatrix b) {
  // dotprod = vector %*% matrix
  vec a2 = as<vec>(a); // vector to 1-by-n matrix
  int n = a.size();
  mat aa;
  aa.insert_cols(0, a2);
  aa.reshape(1, n);

  mat bb = as<mat>(b);
  mat cc = aa * bb;
  return(wrap(cc));
}

// [[Rcpp::export]]
NumericVector m2v(NumericMatrix x){
  // matrix to n-by-1 pseudo-vector
  mat mx = as<mat>(x);
  vec ret = vectorise(mx);
  return(wrap(ret));
}
```

```
// [[Rcpp::export]]
NumericMatrix v2m(NumericVector v, int nrow, int ncol){
  NumericMatrix out(nrow, ncol);
  for (int i = 0; i < v.size(); i++ ){
    out[i] = v[i];
  }
  return out;
}

// [[Rcpp::export]]
NumericMatrix InitialMatrix(int nrow, int ncol, bool fill = false){
  int n = nrow * ncol;
  NumericMatrix ret(nrow, ncol);
  NumericVector::iterator i = ret.begin();
  NumericVector::iterator j;
  NumericVector val;
  if (fill == true) {
    val = rep(NumericVector::create(0.1), n);
  } else {
    val = runif(n);
  }
  for (j = val.begin(); j != val.end(); ++i, ++j) {
    *i = *j;
  }
  return clone(ret);
}

// [[Rcpp::export]]
NumericMatrix dotrans(NumericMatrix x){
  mat xx = as<mat>(x);
  return(wrap(xx.t()));
}

// [[Rcpp::export]]
List RNN_Train(NumericVector A, NumericVector B, bool verbose){
  // network basic parameters
  double alpha = 0.1;
  int input_dim = 2;
  int hidden_dim = 16;
  int output_dim = 1;
  int const bin_dim = 8;

  // network weights; [final output]
  NumericMatrix syn0 = 2 * InitialMatrix(input_dim, hidden_dim) - 1;
  NumericMatrix syn1 = 2 * InitialMatrix(hidden_dim, output_dim) - 1;
  NumericMatrix synh = 2 * InitialMatrix(hidden_dim, hidden_dim) - 1;
//   syn0 = syn0 * 0 + 0.1;
//   syn1 = syn1 * 0 + 0.1;
//   synh = synh * 0 + 0.1;

  NumericMatrix syn0_up = InitialMatrix(input_dim, hidden_dim) * 0;
  NumericMatrix syn1_up = InitialMatrix(hidden_dim, output_dim) * 0;
  NumericMatrix synh_up = InitialMatrix(hidden_dim, hidden_dim) * 0;
```

```
// correct answer
NumericVector C = A + B;

int N = A.size();
for (int smp_i = 0; smp_i < N; smp_i ++) { // iterate each sample
  if (verbose) {
    Rcout << "## Sample: " << smp_i << std::endl;
    Rcout << "syn0(0,0) = " << syn0(0, 0) << std::endl;
  }
  int aint = A[smp_i];
  int bint = B[smp_i];
  int cint = C[smp_i];
  std::bitset<bin_dim> a(aint);
  std::bitset<bin_dim> b(bint);
  std::bitset<bin_dim> c(cint);
  NumericVector cHat(bin_dim); // RNN learning binary digit

  double err_sum = 0.0;

  List l2_deltas;
  List l1_vals;
  l1_vals.push_back(rep(NumericVector::create(0), hidden_dim));

  // FP begins
  if (verbose ) Rcout << "-- FP" << std::endl;
  for (std::size_t i = 0; i < a.size(); ++i) { // iterate time-steps
    // input and output of each time-step
    NumericVector x = NumericVector::create(a[i], // Notice bit-set operator
                                            b[i]); // right-most
    NumericVector y = NumericVector::create(c[i]);
    // hidden_layer ~ input + prev_hidden
    NumericVector l1 = sigmoid(dotprodvm(x, syn0) +
      dotprodvm(as<NumericVector>(l1_vals[l1_vals.size() - 1]), synh));

    // output layer
    NumericVector l2 = sigmoid(dotprodvm(l1, syn1));
    // error at output layer
    NumericVector l2_err = y - l2;
    err_sum += sum(abs(l2_err));
    l2_deltas.push_back(l2_err * sigmoid_deriv(l2));

    if (verbose) {
      Rcout << ">> sample " << smp_i << "pos" << i << "=" << x << std::endl;
      Rcout << a[bin_dim - i - 1] << std::endl;
      Rcout << l2_err << std::endl;
    }

    // save output to be displayed
    cHat[bin_dim -i -1] = round(l2[0]);
    // save hidden layer to be used for next time-step
    l1_vals.push_back(clone(l1));
  }
  // FP ends
```

```
if (verbose) {
  for (List::iterator li = l1_vals.begin(); li != l1_vals.end(); li ++){
    Rcout << "l1 vals: " << as<NumericVector>(*li) << std::endl;
  }
}

// BP begins
if (verbose) Rcout << "-- BP" << std::endl;
// layer-1 at "next-time"-step
NumericVector future_l1_delta = rep(NumericVector::create(0), hidden_dim);
for (std::size_t i = 0; i < a.size(); ++i) {
  NumericVector x = NumericVector::create(a[bin_dim - i - 1],
                                          b[bin_dim - i - 1]);
  NumericVector l1 = l1_vals[l1_vals.size() - i - 1];
  NumericVector l1_prev = l1_vals[l1_vals.size() - i -2];
  // delta at output layer
  NumericVector l2_delta = l2_deltas[l2_deltas.size() -i - 1];
  // delta at hidden layer
  NumericVector l1_delta = (dotprodvm(l2_delta, dotrans(syn1)) +
                            dotprodvm(future_l1_delta, dotrans(synh))) *
                            sigmoid_deriv(l1);
  // collect updates untill all time-steps finished
  syn1_up += dotprodmm(v2m(l1, l1.size(), 1),
                       v2m(l2_delta, 1, l2_delta.size()));
  synh_up += dotprodmm(v2m(l1_prev, l1_prev.size(), 1),
                       v2m(l1_delta, 1, l1_delta.size()));
  syn0_up += dotprodmm(v2m(x, x.size(), 1),
                       v2m(l1_delta, 1, l1_delta.size()));

  future_l1_delta = l1_delta;

  if (verbose){
    Rcout << "bp input at pos: " << i << "=" << x << std::endl;
    Rcout << "future delta: " << future_l1_delta << std::endl;
  }
}
// BP ends here

// Update netowrk parameters
syn0 += (syn0_up * alpha);
syn1 += (syn1_up * alpha);
synh += (synh_up * alpha);

syn0_up = InitialMatrix(input_dim, hidden_dim) * 0;
syn1_up = InitialMatrix(hidden_dim, output_dim) * 0;
synh_up = InitialMatrix(hidden_dim, hidden_dim) * 0;

if (verbose) {
  Rcout << "!!After" << std::endl;
  Rcout << syn0(0, 0) << syn0(0, 1) << syn0(1, 0) << syn0(1, 1) << std::endl;
  Rcout << syn1(0, 0) << syn1(1, 0) << syn1(2, 0) << syn1(0, 3) << std::endl;
  Rcout << synh(0, 0) << synh(0, 1) << synh(1, 0) << synh(1, 1) << std::endl;
  Rcout << syn0_up(1, 2) << std::endl;
```

```
    }

    if (!verbose && smp_i % 1000 == 0 ){
      Rcout << "Sample " << smp_i << std::endl;
      Rcout << "Overall Error: " << err_sum << std::endl;
      Rcout << "Pred: [" << cHat << "]" << std::endl;
      Rcout << "True: " << c.to_string() << std::endl;
      double cHatInt = 0.0;
      for (int i = 0; i < cHat.size(); i ++){
        cHatInt += pow(2.0, cHat.size() - i - 1) * cHat[i];
      }
      Rcout << "Calc Binary: " << std::endl << a << std::endl << b << std::endl;
      Rcout << "Calc Decimal: " << aint << "+" << bint << "=" << cHatInt << std::endl;
      Rcout << "----" << std::endl;
    }

  }

  List syn = List::create(_["syn0"] = syn0,
                          _["syn1"] = syn1,
                          _["synh"] = synh);
  return syn;
}

/*** R
require(Rcpp)
require(RcppArmadillo)
require(RcppEigen)
set.seed(2016)
max_num <- 2^8 -1
n <- 10000
A <- c(sample(1:(max_num/2), n, replace = T), 1)
B <- c(sample(1:(max_num/2), n, replace = T), 1)
vFlag <- F
rnn_fit <- RNN_Train(A, B, vFlag)
*/
```