Advanced Statistical Learning (MATH80619)

Deep Learning in R

Presented to
Denis Larocque
Aurélie Labbe

by

Estefan Apablaza-Arancibia	(11271806)
Adrien Hernandez	(11269225)

Master of Science (M. SC.)

HEC MONTRĒAL

Data Science & Business Analytics
Québec, Canada
March 31, 2020

1 Introduction

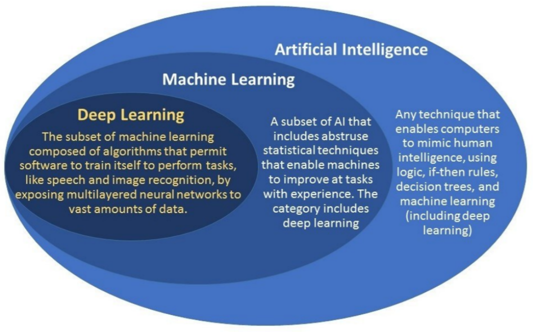
These days, ”deep learning” buzz word is part of multiple repositories projects and companies around the world. Most projects are built around the programming language python. That being said it will be interesting to see if this trend is viable in the R studio environment. To start, a literature review covers the neural networks and deep learner’s algorithms, focusing on different types of neural networks architectures and describing for which kind tasks each neural network architecture has been designed for. Furthermore, the methodology section will try to describe briefly the workflow of deep learning training and predictions, the dataset that could be used for different algorithms and finally some characteristic use to compare different packages. Then, an exhaustive list of the R libraries allowing to build neural networks and deep learners models. Correspondingly, the advantages and disadvantages of each library, their capabilities as well as what you can expect when using them. To sum up, the last section will give concrete examples on how to implement the neural networks and deep learners models with these libraries, using the example data of the methodology section.

2 Litterature Review

2.1 What is Deep Learning and why use it?

2.1.1 Feed Forward Neural Network

Deep learning is a subset field of artificial intelligence and can be seen as a specific way of doing machine learning. Deep learning algorithms can be seen as feature representation learning. Indeed, by applying to the data multiple non-linear transformations through the use of hidden layers, deep learning models have their own trainable feature extraction capability making it possible to learn specific features from the data without needing a specific human domain expert. This means that deep learning models won’t require the features extraction step that is essential for classic machine learning models. However, increasing the model’s capacity by adding hidden layers, requires increasingly computing power and slow down the training process of the model. The choice of hyperparameters, programming languages and memory management will therefore be important criteria to take into account while building deep learning models


Artificial Intelligence
Machine Learning
Deep Learning
The subset of machine learning composed of algorithms that permit software to train itself to perform tasks, like speech and image recognition, by exposing multilayered neural networks to vast amounts of data.
A subset of AI that includes abstruse statistical techniques that enable machines to improve at tasks with experience. The category includes deep learning
Any technique that enables computers to mimic human intelligence, using logic, if-then rules, decision trees, and machine learning (including deep learning)
Figure 1: Artificial Intelligence vs Machine Learning vs Deep Learning [1]

Since the last decade, deep learning models have shown notable predictive power and have been revolutionizing an important number of domains such as computer vision, natural language understanding, fraud detection, health and much more.
As a first glance in the subject, it is highly recommended it to read a reference from pioneers in the field ([11, Chapter 1]).

”Deep feedforward networks, also called feedforward neural networks, or multilayer perceptrons (MLPs), are the quintessential deep learning models.” [11] FNN models are inspired from biology and by the way the human brain works. In a neural network, each neuron takes input from other neurons, processes the information and then transmits outputs to next neurons. Artificial neural networks follow the same process as each neuron will perform the weighted sum of inputs and will add a bias as a degree of freedom. It will then apply a non-linear transformation before outputting the information. Thus, the information goes forward in the network; neurons transmit information from the input layers towards the output layer. It is important to know that in a feedfoward neural networks (fig. 2) the neurons of a same layer are not connected to each other; they do not allow any feedback connections within a same layer. If we want to allow this process, we will be looking at recurrent neural networks.

The equation a neuron input is

$$a(x) = b + \sum_i w_i x_i \tag{1}$$

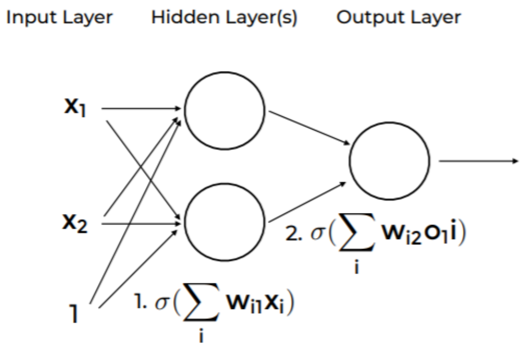
and the output

$$h(x) = g(a(x)) \tag{2}$$

where:

x	= the input data
b	= the bias term
w	= the weight or parameter
$g(...)$	= the activation function

Here, **the bias term b** and **the weights or parameter w** will be learned by the model in order for the minimize a cost function, through the use of the gradient descent method. Then, the network will use the backpropagation algorithm where the error is backpropagated from the output to the input layers and the bias and weighted are then updated accordingly.
Regarding the **activation function g(...)**, the common practice is to use a ReLu (rectified linear unit) as the activation function for the neurons of the hidden layers. Regarding the neurons of the last layer, the activation function will be chosen accordingly to the task we want our model to perform:



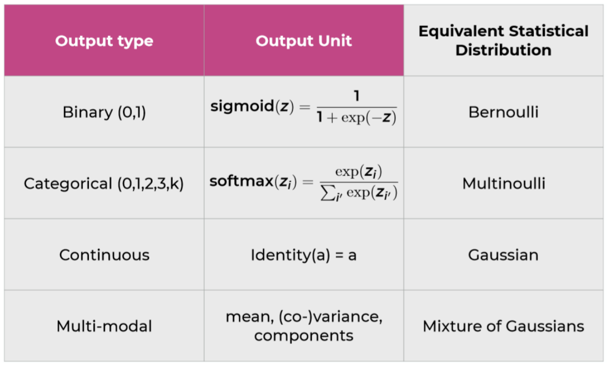
Input Layer
Hidden Layer(s)
Output Layer
x_1
x_2
1
1. $\sigma(\sum_i w_{i1} x_i)$
2. $\sigma(\sum_i w_{i2} o_1 i)$

Figure 2: feedforward neural networks

A detailed explanation of the theory of feedforward neural networks can be found in [11, Chapter 6]

2.1.2 CNN

The CNN are a modified architecture of FNN that leverages the feature engineering that used to be hand made by domain experts. This class of deep neural network are commonly used for image recognition and classification that can serve different applications such as facial recognition, document analysis and speech recognition. The original FNN are not suited analyzing large size images since the weights increase exponentially and, at the end, don't perform very well.



Output type	Output Unit	Equivalent Statistical Distribution
Binary (0,1)	$\text{sigmoid}(z) = \frac{1}{1+\exp(-z)}$	Bernoulli
Categorical (0,1,2,3,k)	$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{i'} \exp(z_{i'})}$	Multinoulli
Continuous	Identity(a) = a	Gaussian
Multi-modal	mean, (co-)variance, components	Mixture of Gaussians

Figure 3: Activation functions: output units. [8, slide 24]

A standard architecture of a CNN model is commonly represented this way:

- We start with an input image to which we are going to apply several kernels, a.k.a features detectors, in order to create feature maps that will form what we call a convolutional layer.
- A common practice is to apply the ReLu activation function to the convolutional layer output in order to increase the non-linearity of the images.
- Then we use the pooling method to create a pooling layer.
- The next step will be to combine all the pooled images from the pooling layer into one single vector, which is called flattening and will be the input of a FNN that will perform the desired task, for instance, classification.

CNN parameters are trained with the **backpropagation** algorithm where the gradients are propagated from the output layer of the FFN to the input of the CNN in order to minimize a cost function, most of the time a categorical cross-entropy if we are performing a multi-class classification.

To get a detailed explanation of convolutional neural networks we recommend to read [11, Chapter 9].

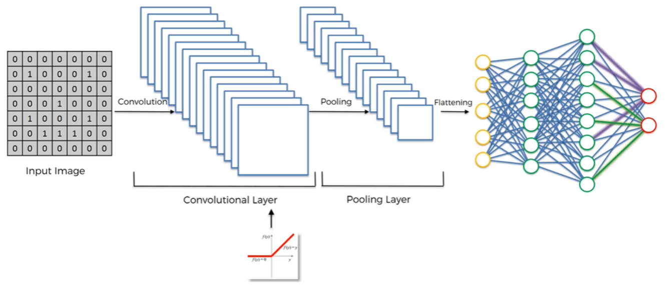


Figure 4: CNN, Deep Learning A-Z by SuperDataScience

2.1.3 RNN

Recurrent neural networks are a more advanced type of deep learning architecture having proven state-of-art performance for solving tasks related to sequential data such as Natural Language Processing (NLP), anomaly detection and event forecasting. As a key differentiator from feedfoward neural networks is that RNNs use feedback loops connections instead of feedforward connections and get an internal memory. Indeed, that take as input each step of the sequential data as well as what they have learned over time. One of their other advantage is to be able to handle input and output sequences with different sizes. ”A recurrent neural network can be thought of as multiple copies of the same network, each passing a message to a successor”[13].

A standard architecture of an RNN model is commonly represented this way:

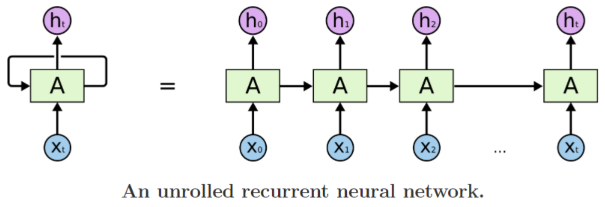


Figure 5: RNN, Understanding LSTM Networks by Olah

- The training uses the input x as part of the data sequence or vector. For example, if our data is **x = ”I am learning deep learning”**, then our x_1 will be ”I”, and our x_2 will be ”am”, etc.
- We will also use a second input which is the hidden state of the previous cell of our network. This means that in this network our neurons are connected by a feedback loop. This will allow our network to capture the ”temporal” information of our data and gives it a memory.

RNN parameters are trained with the **backpropagation through time** algorithm where the gradients are propagated from the output layer of the RNN in order to minimize a cost function, most of the time a categorical cross-entropy if we are performing a multi-class classification, for example. However, a simple RNN network will suffer of gradient vanishing or exploding because the sequence lengths. State-of-art algorithms such as GRU or LSTM exists to tackle this problem.

A detailed explanation of recurrent neural networks can be found in the [11, Chapter 10]. We also recommend reading this blog post [13].

2.2 Deep Learning integration in R

The integration of deep learning in R can be separate in two parts ; (1) The API integration and (2) the standalone R packages.

1. The API will give the possibility to control externally through R an existing installation. In other words, a software translator for an already-known software installation. This approach is not always trivial to install nor to manipulate but in the long term will probably give the best flexibility in terms of Deep Learning projects.
2. The standalone R packages will be packages that require no third party software in order to create the deep learning projects. This approach is relatively fast to install but is restrained in terms deep learning architecture.

A list of the available resources with installation and examples tutorials can be found in section **??** and **??** .

3 Methodology

This section is divided in two; the dataset description and the benchmark type. The dataset description is mostly to describe what dataset is going to be used for the different type of deep learning algorithms. The second section will describe two indicators in order to compare results of similar algorithms. The first indicator is mostly about a simple accuracy based on the same training and validation dataset and second indicator is about the time it took to execute the same task on different resources.

3.1 Workflow

The workflow of a deep learning project in R we recommend to follow is described below:

- Collection, pre-processing and cleaning of the data in order to make them ready to train the model. Partition the data into a training set and a validation/test set (most of the time 80% of the data goes to training and 20% goes to testing). You can use cross-validation if you don't have enough data, but it shouldn't be the case in most of the deep learning projects.
- Choose the right model according to the task you want to perform. Some models have been especially created in order to perform well on a specific task, for example, CNN have been created to solve images classification-related tasks.
- Train the selected models with your training data by checking at the same time how your validation accuracy (or another metric that corresponds the best to your task) is evolving through the training process.
- Modify your models' hyperparameters and experiment changes in the model architecture in order to improve your model accuracy.
- Evaluate and compare your models on the test data.

3.2 Dataset

3.2.1 FNN

For the FNN, the training data will be the well known *Iris* dataset. It contains four features (Length and Width of Sepal and Petals) and one categorical target variable representing 3 type iris species ((*Iris setosa*, *Iris virginica* and *Iris versicolor*)). This dataset of 50 observations is pretty well known and is perfect to show a small example of FNN.

3.2.2 CNN

For the CNN, the training data will also be a well-known dataset called CIFAR-10. The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. The easiest way in R to download this package is using the Keras API. The training dataset has 50 000 images in three colours (Red-Green-Blue) that are 32 pixels by 32 pixels. The testing dataset has a total of 10 000 images

3.2.3 RNN

The dataset used to test an RNN is the IMDB Movie reviews for sentiment classification. The reason to use this dataset is that Keras API also offers a complete bundle (of 25 000 reviews for training and another 25 000 for testing) with multiple parameters to help you get started with sentiment classification using RNN. Each review is classified as "1" (good) or "0" (bad) and the word index is already done. The total word vocabulary is 88 000 words.

3.3 Characteristic

3.3.1 Accuracy

As the purpose of this tutorial is to learn to build the architecture of neural networks in R, we will not focus on accuracy measurement.However, it might come handy for users to have a tool or starting point therefore an extensive reading of this source [3] is highly recommended. Since for all three algorithms (FNN , CNN and RNN) it is mostly a classification models the accuracy will be $1 - classification_{error}$.

3.3.2 Time elapsed

Sometime the training timing can be a deal breaker for some users. This characteristic is calculated from a simple R package ,that being said it is really important to not generalized the tutorials results. It is simply to give an idea about the training time since it depends on multiple factors (e.g., computer characteristics such as memory , CPU, etc.).

4 Available resources

As mentioned in the section 2.2, we will firstly introduce the most important R packages coming from the CRAN platform for deep learning. Then, we will give an overview of the different API packages available. For each R package (not API based), we give a description as well as a visual example of how to build a neural network model with it. Moreover, we highlight the most important Pros and Cons, although we highly suggest you to read the package's official documentation that we also provide.

4.1 R packages

4.1.1 Neuralnet package

Description According to its CRAN description, the package allows the "training of neural networks using the backpropagation, resilient backpropagation with (Riedmiller, 1994) or without weight backtracking (Riedmiller, 1993) or the modified globally convergent version by Anastasiadis et al. (2005). The package allows flexible settings through custom choice of error and activation function. Furthermore, the calculation of generalized weights (Intrator O & Intrator N, 1993) is implemented." This package uses $C/C++$ in backend. More information can be found on the package source [19].

Important default parameters of a FNN model with the neuralnet package:

```
neuralnet(formula,
      data = Y~X1+...+Xn,
      hidden = 1,
      threshold = 0.01,
      stepmax = 1e+05,
      rep = 1,
      startweights = NULL,
      learningrate.limit = NULL,
      algorithm = "rprop+",
      err.fct = "sse",
      act.fct = "logistic",
      linear.output = TRUE)
```

(a) Training function

```
#' @param formula : a symbolic description of
#          the model to be fitted.
#' @param data : a data frame containing the
#          variables specified in formula.
#' @param hidden : a vector of integers
#          specifying the number of hidden
#          neurons (vertices) in each layer.
#' @param threshold : a numeric valuethe
#          threshold for the partial
#          derivatives of the error function
#          as stopping criteria.
#' @param stepmax : the maximum steps for
#          the training of the neural network.
#' @param rep : the number of repetitions
#          for the neural network's training.
#' @param algorithm : a string containing
#          the algorithm type to calculate the
#          neural network.
#' @param err.fct : a differentiable function
#          that is used for the calculation of
#          the error
#' @param act.fct : a differentiable function
#          that is #used for smoothing the result
#          of the cross product of the covariate
#          or neurons and the weights.
#' @param linear.output :  If act.fct should not
#          be applied to the output neurons set
#          linear output to TRUE, otherwise to FALSE.
#' @return a neuralnet model
```

(b) Parameters

Figure 6: Neuralnet package

Pros

- Very easy to use and to build a quick FFN and deep FNN model.
- Allows to use several types of backpropagation algorithms. By default, the resilient backpropagation algorithm is used "rprop+" instead of regular backpropagation which is an algorithm not very used in practice as it is trickier to implement. We can, however, decided to change the rprop+ for a standard backpropagation by changing for **algorithm = "backprop"**.
- Several hidden layers and neurons per layer can be added. By default, the algorithm uses only 1 hidden layer with 1 neuron, but it can be increased by adding the command line: **hidden = c(3,3)** to get 2 hidden layers of 3 neurons each.
- Allows to easily plot and visualize the model and its parameters with the command line: **plot(model).**
- Allows to use custom activation functions.

Cons

- One of the disadvantages of this package is that it requires some data pre-processing as it only works with numeric inputs. Therefore, factor variables will need to be transformed into dummies during the pre-processing phase.

- The package doesn't provide built-in normalization functions. Therefore, it is recommended to manually normalize the data before using them as input in the neural networks in order to reduce the amount of iteration of the algorithm.

4.1.2 nnet package

Description This package comes from the CRAN platform and allows to build and fit, ”FNN with a single hidden layer and multinomial log-linear models”. This package uses C/C++ in backend. More information can be found on the package source [16].

Important default parameters of a FNN model with the neuralnet package:

```
nnet(X, Y,
     size,
     linout=FALSE,
     entropy=FALSE,
     softmax=FALSE,
     maxit = 100,
     decay = 0)
```

(a) Training function

```
#' @param X : x values for examples
#' @param Y : target values for examples
#' @param size : number of units in the hidden
#         layer
#' @param linout : switch for linear output
#         units
#' @param entropy : switch for entropy
#' @param softmax : switch for softmax
#' @param maxit : maximum number of iterations
#' @param decay : parameter for weight decay
#' @return a nnet model
```

(b) Parameters

Figure 7: nnet package

Pros

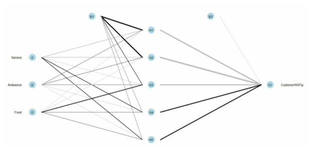
- One of the easiest R packages to build a quick FFN model.

Cons

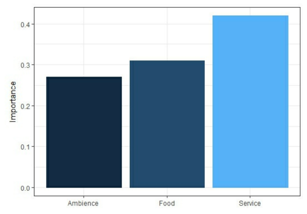
- It does not offer a lot of flexibility, and can only apply logistic sigmoid function for the hidden layer activation and cannot use tanH and ReLu.
- This package does not allow to use more than one hidden layer, and does not have any feature to find the optimal number of neurones in the hidden layer. It is up to the analyst to build a loop to test by cross-validation, for example, the optimal hyperparameter values.
- Cannot use classical backpropagation algorithm to train the network. It only uses the BFGS (Broyden-Fletcher-Goldfarb-Shanno) which is a quasi-Newton method and therefore increases the number of computations to reach a local optimal. However, applied on a small dataset with a model having only one hidden layer does not seem to be a problem.
- Does not have any function to plot the model.

4.1.3 NeuralNetTools package

Description This package is a complement to R's neural networks packages as it allows to visualize and perform analysis on neural network models. ”Functions are available for ploting, quantifying variable importance, conducting a sensitivity analysis, and obtaining a simple list of model weights.” More information can be found on the package source [7].



(a) NeuralNetTools, plotnet function



(b) NeuralNetTools, Input variables importance

Figure 8: Embedding dimension hyperparameter [10]

Pros

- Very easy to plot your neural network models with the function **plotnet(model)**.
- Visualize the input variables importance to the output prediction with the Garson and olden algorithm.
- Perform sensitivity analysis on your neural networks model using the Lek profile method.
- Works with the models built from different packages: **caret, neuralnet, nnet, RSNSS**
- Can plot neural networks with pruned connections from the RSNSS package.

Cons

- Does not provide any function for neural network model development.
- Is not optimized to visualize large neural networks models.

4.1.4 Deepnet package

Description This package is available on the CRAN platform and has been written specifically written for R. It allows to "implement some deep learning architectures and neural network algorithms, including backpropagation, Restricted Boltzmann Machine (RBM), Deep Belief Network (DBN) and Deep autoencoder". More information can be found on the package source [18].
Important default parameters of a deep FNN model with weights initialized by DBN:

```
dbn.dnn.train(X, Y, hidden=c(5),
              activationfun="sigm",
              learningrate=0.5,
              momentum=0.5,
              learningrate_scale=1,
              output="softmax",
              numepochs = 3,
              batchsize = 100,
              hidden_dropout = 0,
              visible_dropout = 0,
              )
```

(a) Training function

```
#' @param X : matrix of x values for examples
#' @param Y : vector or matrix of target values
#         for examples
#' @param hidden : vector for number of units
#         of hidden layers
#' @param activationfun : activation function
#         of hidden unit
#' @param learningrate : learning rate for
#         gradient descent
#' @param momentum : momentum for gradient descent
#' @param learningrate_scale :learning rate will be
#' @param output : function of output unit, can be
#         "sigm","linear" or "softmax".
#' @param numepochs : number of iteration
#         for samples
#' @param batchsize : size of mini-batch
#' @param hidden_dropout : drop out fraction for
#         hidden layer
#' @param visible_dropout : drop out fraction
#         for input layer
#' @return a dbn model
```

(b) Parameters

Figure 9: Deepnet

Pros

- Allows to load benchmark datasets such as MNIST with the function load.mnist(dir)
- Allows to initialize the weights of a FNN with the Deep Belief Network algorithm (dbn.dnn.train()) or Stacked AutoEncoder algorithm (sae.dnn.train()).
- Allows to have an estimation of the probabilistic distribution of a dataset through the use of the Restricted Boltzmann machine algorithm.
- Have more activation functions than the other R packages. Hidden Layers activation function can be linear, sigmoid or tanh. The output activation function can be linear, sigmoid or softmax.

Cons

- Does not support the ReLu activation function for hidden layers.
- Not the fastest package due to its implementation in R.
- Does not provide a lot of hyperparameters tuning options, and is not user-friendly.

4.1.5 brnn package

Description This package available on the platform CRAN, allows to perform ”Baeysian regularized neural networks including both additive and dominance effects, and allows to take advantage of multi-core architectures via a parallel computing approach using openMP for the computations” [14] therefore the package is written in R but uses the API OpenMP for faster calculation. More information can be found on the package source [17]
Important default parameters of the brnn function:

```
brnn(formula=  y ~ x1 + x2,
      neurons=2,
      normalize=TRUE,
      epochs=1000,
      mu=0.005,
      mu_dec=0.1,
      mu_inc=10,
      mu_max=1e10,
      min_grad=1e-10,
      change = 0.001,
      cores=1,
      verbose=FALSE
     )
```

(a) Training function

```
#' @param formula : A formula of the form
#          y ~ x1 + x2
#' @param neurons : the number of neurons
#' @param normalize : normalize inputs and output
#' @param epochs : maximum number of epochs
#' @param mu : positive number that controls
#          the behaviour of the Gauss-Newton
#          optimization
#' @param mu_dec : the mu decrease ratio
#' @param mu_inc : the mu increase ratio
#' @param mu_max : maximum mu before training
#          is stopped
#' @param min_grad : minimum gradient
#' @param change : The program will stop
#          if the maximum
#' @param cores : Number of cpu cores to
#          use for calculations
#' @param verbose : print iteration history
#' @return a brnn model
```

(b) Parameters

Figure 10: nnet package

Pros

- Allows to add Additive and Dominance effects in the neural networks models.
- Take advantage of multicore processors (only for UNIX-like systems).
- Allows to use a Gauss-Newton algorithm for optimization.
- The package is able to assign initial weights by using the Nguyen and Widrow algorithm.
- Include an algorithm to deal with ordinal data by using the function **brnn_ordinal(x, ...)**
- It has a function to normalize and unnormalized the data.

Cons

- Cannot use the classic backpropagation algorithm for optimization.
- The package fits a two layers neural networks and it is not possible to increase the number of hidden layers.

4.1.6 rnn package

Description Avaible from the CRAN platform, this package allows the "implementation of a Recurrent Neural Network architecture in native R, including Long-Short Term Memory (LSTM), Gated Recurrent Unit (GRU) and vanilla RNN." More information can be found on the package source [15].

Important default parameters of the rnn function:

```
# train the model
model <- trainr(Y=data_y,
                X=data_x,
                learningrate = 0.1,
                hidden_dim = 10,
                batch_size = 100,
                numepochs = 10)
```

(a) Training function

```
#' @param X : array of input values
#' @param Y : array of output values
#' @param learningrate : learning rate to be
#          applied for weight iteration
#' @param hidden_dim : dimension of hidden layer
#' @param batch_size : number of samples used at
#          each weight iteration
#' @param numepochs : number of iteration
#' @return a rnn model
```

(b) Parameters

Figure 11: rnn package

Pros

- Allows to run a demonstration of how RNN models work by using predefined values and allowing the user to see the impact of a change in the model's hyperparameters by using the function **run.rnn_demo()**
- Allows to plot the model's errors through all epochs.
- Allows to use LSTM and GRU models.

Cons

- Uses R native, therefore it might not be the best package in terms of time computation.

4.2 API packages

4.2.1 Keras

Description This API is really popular in the Python world because it acts like a wrapper of more popular libraries. It might become hard to follow but Keras has multiple backends such as Tensorflow, Theano and MXNET. The *Keras* is mostly used to link the wrapper tool with R. The most useful reference one can get for this workflow is written by two major players ; *Keras* creator and *RStudio* creator. It explains in detail the best practices in order to use *Keras* in R [9]. More information can be found on the package source [5].

Pros

- Give a lot of flexibility since can basically be used to create all types of deep learning algorithms.
- Easy and fast prototyping
- Allows to use ReLu activation function which is not available in R packages from CRAN.
- Construct models exactly the same way as in python with a minimalist philosophy.
- Give the possibility to save everything on the workspace (model, training history, etc.)

Cons

- It is an encapsulation of a program inside a python environment.
- For some, it might be to high level and limit the customization.
- It can only work on top Tensorflow, CNTK and Theano backend.
- Not able to set a seed to get reproducible results as it has to be controlled in keras for python running in backend.

4.2.2 Tensorflow

Description This library was created by Google and written in $C++$ and python. It is pretty well known in the python environment. Recently, a R package was created to interface this popular library ."'TensorFlow' was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research." More information can be found on the package source [6].

Pros

- Well documented with a lot of tutorials online.
- Pre-trained models accessible

Cons

- Struggles with poor results for speed.
- Not a trivial since consider a low-level coding

4.2.3 MXNet

Description MXNet is an open source Deep Learning framework created by Apache. The framework is mostly backed by Intel, Microsoft and MIT. It offers multiple features and capabilities and is also available in multiple programming languages. The core application is written in C++. It was created with the intention of being scalable and distributed in the cloud. There is a manual but isn't part of the CRAN database. More information can be found on the package source [20].

Pros

- Allows the use of multiple CPUs and multiple GPUs, but it requires to download the package Rtools and a c++ compiler.
- Very easy to use and has a flexible implementation of different neural networks architectures and models.
- The models can be built layer per layer.
- Provide details and information about the learning progress during the training phase.

Cons

- It has a smaller community compared to other popular framework.
- A package more use in the industrial projects and not so much in the research community.
- No built-in function for data/tensor manipulation

4.2.4 rTorch

Description The rTorch CRAN package is to interface the popular open source machine learning library PyTorch. It was developed by Facebook AI Research lab (FAIR). In reality, this provides access to famous python libraries such as numpy (as a method called np) or torchvision. More information can be found on the package source [4].

Pros

- Large number of mathematical functions to manipulate tensors.
- Access to popular python libraries (numpy and torchvision) through the package

Cons

- Doesn't have training capabilities since it is only for manipulations.
- Limited number of concrete examples.

4.2.5 h2o

Description H2o is a "scalable open source machine learning platform that offers parallelized implementations of many supervised, unsupervised and fully automatic machine learning algorithms on clusters". This package allows to run H2o via its REST API through R and offers several advantages such as the ability to show the remaining computation time when a model is running. More information can be found on the package source [12].

```
h2o.deeplearning(X = x,                              #' @param X,
                 Y = y,                              #' @param Y
                 training_frame = train_data,        #' @param training_frame=2
                 epochs = no_epoch,                  #' @param epochs=TRUE
                 seed=123)                           #' @return a h2o connections to the model
```

(a) Training function (b) Parameters

Figure 12: h2o package

Pros

- Very easy to use and includes a cross-validation feature and functions for grid search in order to optimize the model's hyperparameters.
- Allows the use of multiple CPU.
- Provide an adaptive learning rate (ADADELTA) which improves the optimization process by having a different learning rate for each neuron.
- Provide details and information about the learning progress during the training phase.
- Really fast computation training

Cons

- Requires the latest version of Java.
- The deep learning package has a huge number of parameters, however, it doesn't give all the capability of other resources.

5 Examples

In order to give tangible deep learning examples multiple resources were selected for tutorials such as Neuralnet, Keras using Tensorflow backend, H2O and MXNET. However, not all the packages presented in the section 4 were tested. Some these packages didn't have a strong community and were not trivial to implement.

5.1 Neuralnet

5.1.1 Installation

Neuralnet is available on the CRAN and can be easily downloaded and loaded as follow:

```
install.packages("neuralnet")
library(neuralnet)
```

```r
# We will load the data from the library dataset
install.packages("dataset")
library(dataset)
```

Then we split our data into a training set and a test set.

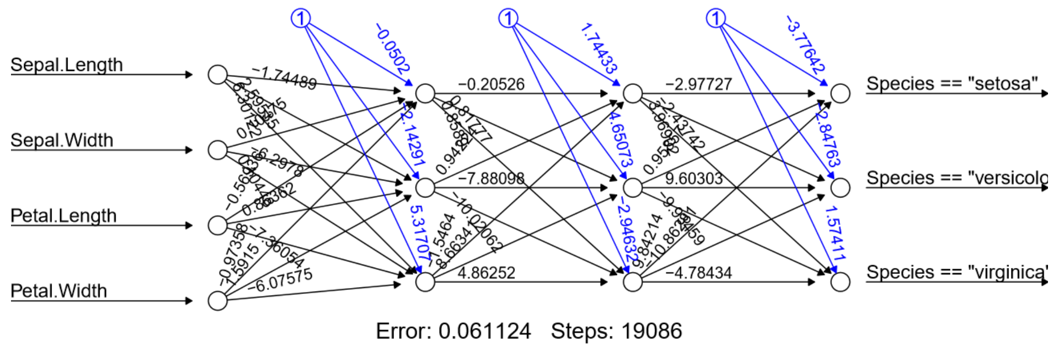
```r
# 70% of the data will be used to train the model
# 30% of the data will be used to test the model performance
n=nrow(iris)
size.train=floor(n*0.7); size.test=floor(n*0.3)

# We use this seed to be able to get the same training and test set everytime
set.seed(123)
# Definition of the observations ID assigned to the train and test data
id.train=sample(1:n,size.train,replace=FALSE)
id.test=sample(setdiff(1:n,id.train),size.test,replace=FALSE)

# We create the training and test dataset
iris_train=iris[id.train,]; iris_test=iris[id.test,]
```

Then, we will build a deep FNN with 2 hidden layers of 3 neurons. As hyperparameters, we will use only 1 epoch, we will use the standard backpropagation algorithm and therefore the learning rate has to be specified. We will use a logistic activation function.

```r
neuralnet_model <- neuralnet((Species=="setosa") +
                               (Species=="versicolor") +
                               (Species=="virginica") ~
                               Sepal.Length+Sepal.Width+
                               Petal.Length+Petal.Width,
                             rep = 1, data = iris_train,
                             algorithm = "backprop",
                             learningrate = 0.01,
                             linear.output = FALSE, hidden = c(3, 3),
                             stepmax = 1000000, act.fct = "logistic")
```

We can easily plot our model to visualize its architecture:

```r
plot(neuralnet_model)
```



The model visualization allows to see if our model corresponds to the hyperparameters we used during training. However, this is not optimized for models that use a high number of hidden layers and neurons.

Finally, to see our model performance on the test dataset, we can use the **predict** function.

```
neuralnet_prediction <- predict(neuralnet_model, iris_test)
table(iris_test$Species, apply(neuralnet_prediction, 1, which.max))
```

```
##
##                1  2  3
##   setosa      14  0  0
##   versicolor   0 17  1
##   virginica    0  0 13
```

The accuracy of this resources is

```
## [1] 0.9777778
```

and the time it took to train is :

```
## Time difference of 9.78815 secs
```

5.2 Keras

5.2.1 Installation

First step, is the installation and download of the Keras files from GitHub :

```
devtools::install_github("rstudio/keras")
```

Then, we need to to install the package and import in the project :

```
library(keras)
install_keras()
```

When the "*Installation complete.*" message appear you have a complete installation with CPU configure on the TensorFlow backend. If you want to take advantage of your GPU (Ensure that you have the hardware prerequisites) you will need to execute a different command as follows:

```
install_keras(tensorflow = "gpu")
```

5.2.2 FNN

First of all, we need to load the data and in order to do so we will use the datasets packages. We the data is in the workspace it will be divided into the train and test.

```
library(datasets)
# Download the Iris data to workspace
data(iris)
# Change the data Iris output for string
iris$Species = sapply(as.character(iris$Species),
                      switch, "setosa" = 1,
                      "versicolor" = 2,
                      "virginica" = 3,
                      USE.NAMES = F)
# Here is another way, to split the data to
spec = c(train = .7,test = .3)
# Set a seed in order to be repreductible
set.seed(123)
# Sample through the dataframe using the sample and cut.
# The variable "g" returns a list of rows for train and
# test
g = sample(cut(seq(nrow(iris)),
               nrow(iris)*cumsum(c(0,spec)),
               labels = names(spec)
               )
           )
# Use the data and the row information to select rows
data_df = split(iris, g)
# Create vector that will contain X (Features variable)
# and Y the target variable
X = c()
Y = c()
X$train = as.matrix(data_df$train[,-ncol(data_df$train)])
X$test = as.matrix(data_df$test[,-ncol(data_df$test)])
Y$train = as.matrix(data_df$train[,ncol(data_df$train)])
Y$test = as.matrix(data_df$test[,ncol(data_df$test)],)
```

When the data is ready, it is time to create the FFN. As an analogy, the keras model should be constructed like a Lego toy game. It starts from the input and goes all the way to the output.

```
# load the library
library(keras)
# Create a sequential model
model_keras = keras_model_sequential()
# Start assembling the model with the first layer as the input.
# A softmax was use in the output because of the
model_keras %>%
  layer_dense(units = 8, activation = 'relu', input_shape = c(4)) %>%
  layer_dense(units = 4, activation = 'softmax')
# Compile model
```

Here is a quick evaluation of the model created.

```
summary(model_keras)
Model: "sequential"

________________________________________________________
Layer (type)               Output Shape          Param #
========================================================
dense_1 (Dense)            (None, 8)             40

________________________________________________________
dense_2 (Dense)            (None, 4)             36
========================================================
Total params: 76
Trainable params: 76
Non-trainable params: 0

________________________________________________________
```

The next step, is the compilation. In this example, it is important to keep in mind that not all the parameters have been use. The loss function, the optimizer and the metrics are the most basic characteristic of an FNN.

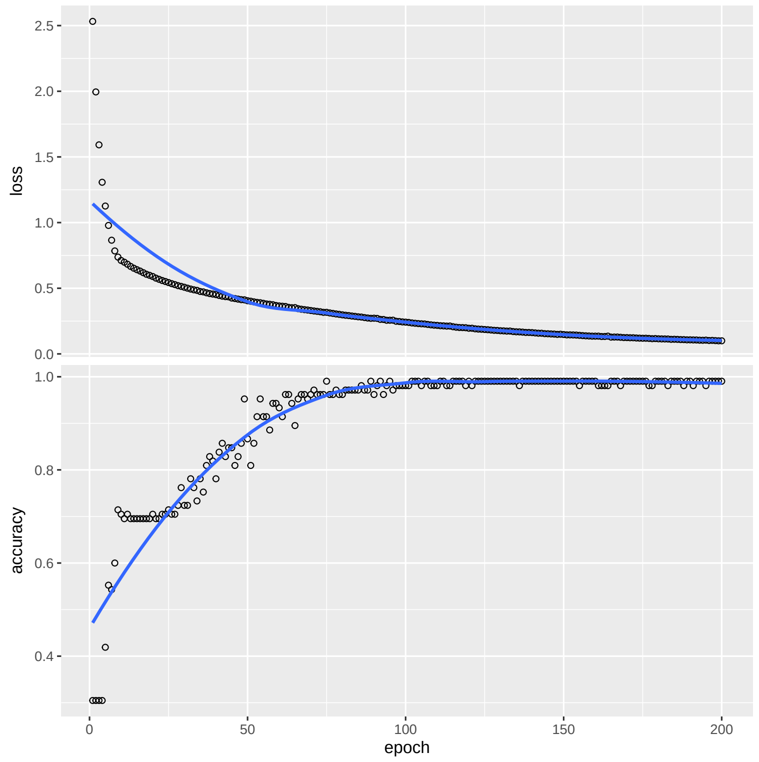
```
model_keras %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_adam(),
  metrics = c('accuracy')
)
```

The next step is really important because it is the training part.

```
history <- model_keras %>% fit(
  as.matrix(X$train),to_categorical(Y$train),
   epochs=200, batch_size=5)
```

This package of the possibility to store the training history :

```
plot(history)

## `geom_smooth()` using formula 'y ~ x'
```



The final step is just to provide an accuracy of the model.

```
preds = predict(model_keras,as.array(X$test))
pred.label.keras <- max.col((preds)) - 1
1-mean(pred.label.keras != as.vector(data_df$test$Species))
```

The accuracy of this resources is

```
[1] 0.9555556
```

and the time it took to train is :

```
## Time difference of 1.213505 mins
```

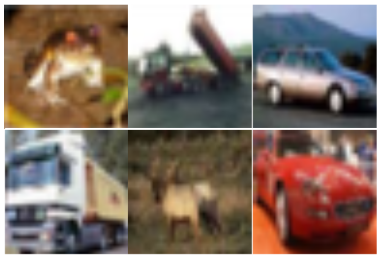
5.2.3 CNN

The first step consists of loading the CIFAR-10 dataset and creating a train and a test set.

```
library(keras)
# We set the seed to get all the outputs reproductible.
set.seed(123)
cifar_10 = dataset_cifar10()
# RGB values are usually encoded between 0 and 255.
# A good practice is to scale them to a value from 0 and 1 b dividing the RGB values by 255.
X_train <- cifar_10$train$x/255
X_test <- cifar_10$test$x/255

# cifar_10 class labels ranging from 0 to 9 are downloaded as integer
# We will use the keras function "to_categorical" to encode them as one-hot.
Y_train <- to_categorical(cifar_10$train$y, num_classes = 10)
Y_test <- to_categorical(cifar_10$test$y, num_classes = 10)
```

To get an understanding of our data we can plot the 6 first images with a for loop.

```
par(mfcol=c(2,3))
par(mar=c(0, 0, 1, 0), xaxs = 'i', yaxs='i')
for (i in 1:6) {
  plot(as.raster(X_train[i,,,]))
}
```



We can now start to implement our CNN model!

```
# Model implementation
model <- keras_model_sequential()

model %>%
  # We start with a first 2D convolutional layer, having a kernel of size 3x3.
  # We use padding = same, meaning that our output tensor will have the same dimensions as our
  #input tensor.
  # Our input_shape is the dimension of each of our image. Here we have a 32x32x3 image (RGB).
  # For black and white images use only 1 dimension (e.g. 32x32x1)
  layer_conv_2d(filter = 32, kernel_size = c(3,3), padding = "same",input_shape = c(32, 32, 3)) %>%
  layer_activation("relu") %>%
  layer_batch_normalization() %>%

  # We use a 2nd convolutional layer where we increase the number of kernel by 2 (32 -> 64)
  layer_conv_2d(filter = 64, kernel_size = c(3,3)) %>%
  layer_activation("relu") %>%
  layer_batch_normalization() %>%

  # We then use a maxpooling layer in order to reduce the dimentionnality of the
  #convolutional layer.
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(0.25) %>%
  # We then flatten the maxpooling layer into a vector that will be feed into a FNN.
  # We set the first layer of our FNN to have 256 hidden neurons.
  layer_flatten() %>%
  layer_dense(256) %>%
  layer_activation("relu") %>%
  #layer_batch_normalization() %>%
  layer_dropout(0.5) %>%

  # We set the output layer of our FNN to have 10 neurons, one for each of the 10 class we
  #want to predict. We use softmax in order to scale all the output values betwen 0 and 1,
  #and that the sum of all of them is equal to 1.
  layer_dense(10) %>%
  layer_activation("softmax")
```

Our model will have 3,709,002 parameters and can be visualized:

```
summary(model)

Model: "sequential_12"
________________________________________________________________________________
Layer (type)                        Output Shape                    Param #
================================================================================
conv2d_54 (Conv2D)                  (None, 32, 32, 32)              896
________________________________________________________________________________
activation_76 (Activation)          (None, 32, 32, 32)              0
________________________________________________________________________________
batch_normalization_52 (BatchNorma (None, 32, 32, 32)              128
________________________________________________________________________________
conv2d_55 (Conv2D)                  (None, 30, 30, 64)              18496
________________________________________________________________________________
activation_77 (Activation)          (None, 30, 30, 64)              0
________________________________________________________________________________
batch_normalization_53 (BatchNorma (None, 30, 30, 64)              256
________________________________________________________________________________
max_pooling2d_26 (MaxPooling2D)     (None, 15, 15, 64)              0
________________________________________________________________________________
dropout_38 (Dropout)                (None, 15, 15, 64)              0
________________________________________________________________________________
flatten_12 (Flatten)                (None, 14400)                   0
________________________________________________________________________________
dense_24 (Dense)                    (None, 256)                     3686656
________________________________________________________________________________
activation_78 (Activation)          (None, 256)                     0
________________________________________________________________________________
dropout_39 (Dropout)                (None, 256)                     0
________________________________________________________________________________
dense_25 (Dense)                    (None, 10)                      2570
________________________________________________________________________________
activation_79 (Activation)          (None, 10)                      0
================================================================================
Total params: 3,709,002
Trainable params: 3,708,810
Non-trainable params: 192

________________________________________________________________________________
```

We then set our hyperparameters to use RMSprop as the optimizater with a learning rate of 1e-3 and a decay of 1e-6. We set our batch size to 64 to train our model with 64 images per iteration. We set epoch to 10, our model will go through the entire training set 5 times. 20% of the training set will be used as validation set, in order to tune the parameters without bias.

```
opt <- optimizer_rmsprop(lr = 1e-3, decay = 1e-6)

batch_size <- 64
epochs <- 5
validation <- 0.2
```

We will us as loss function the categorical_crossentropy that works very well with our softmax activation function in order to perform multi-label classification. We will look at the accuracy as the final metric to check the performance of our model on the test dataset.

```
model %>%
    compile(loss = "categorical_crossentropy",
            optimizer = opt, metrics = "accuracy")
```

Now, we will train our model, and we will store it into a list called **history**.

```
start_time <- Sys.time()
history_cnn_keras <- model %>% fit(
    X_train, Y_train,
    batch_size = batch_size,
    epochs = epochs,
    validation_split = validation,
    shuffle = TRUE)
end_time <- Sys.time()
time_cnn_keras = end_time - start_time
```

By default keras plot our model loss and accuracy per epoch on the training set and on the validation set.

Then, we will evaluate our model performance on our test set.

```
model %>% evaluate(X_test, Y_test,verbose = 0)

$loss
[1] 1.70912

$accuracy
[1] 0.598
```

We can see that our accuracy is 59.8%.
And the time it took to train is:

```
model %>% evaluate(X_test, Y_test,verbose = 0)
Time difference of 19.25289 mins
```

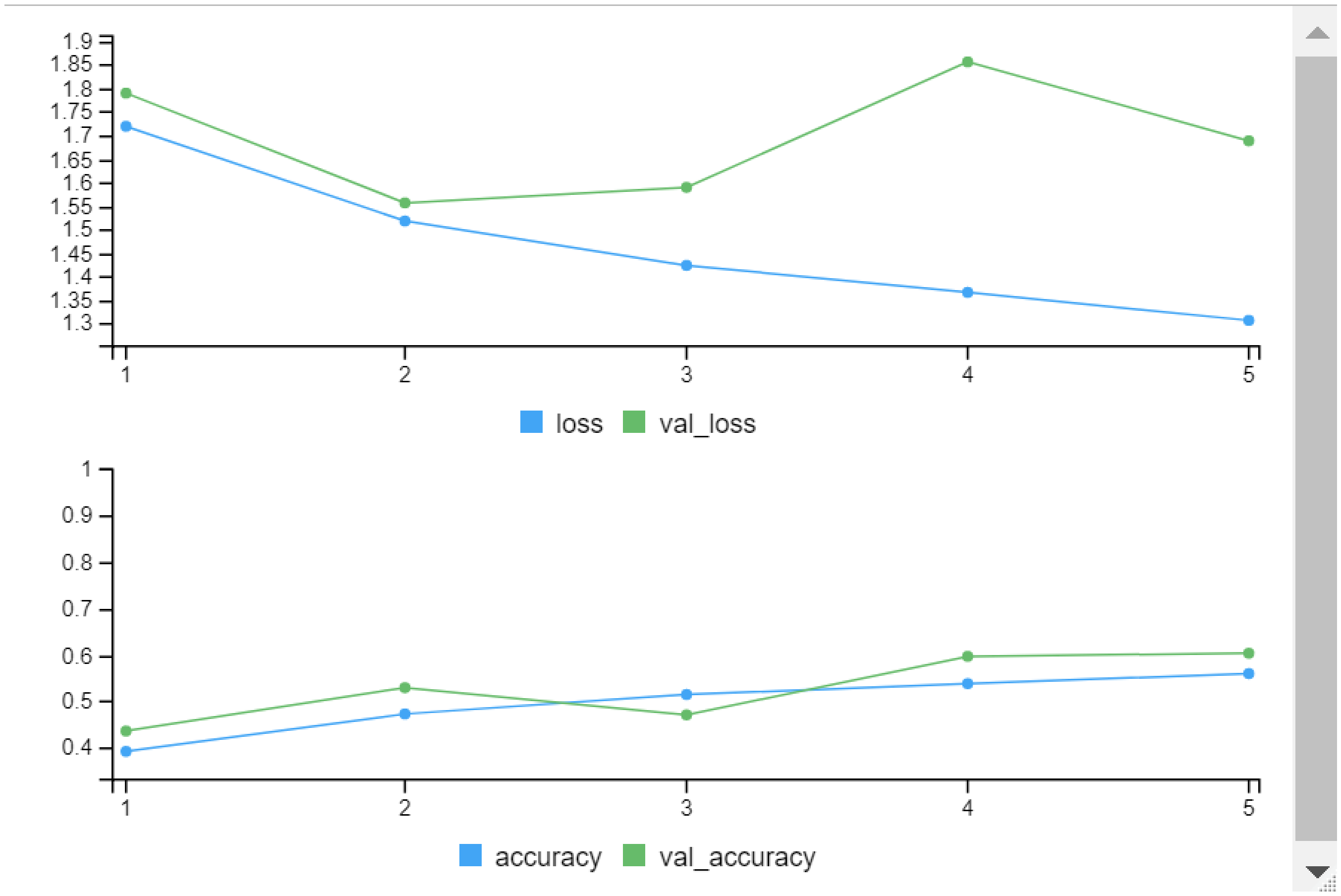
In order to predict with our model on a new dataset, we can use Keras **predict_classes** function.

```
Y_predicted <- model %>% predict_classes(X_test)
```

5.2.4 RNN

This is an example that the keras package uses for text classification. However it is not an RNN or in this case an LSTM algorithm.

As a first step, it is important to download the data from keras.



```r
library(keras)
library(dplyr)
library(ggplot2)
library(purrr)
word_index <- dataset_imdb_word_index()
length(word_index)
# This is in order to limit the number of words
max_features = length(word_index)

# cut texts after this number of words (among top max_features most common words)
maxlen = 45
```

Unfortunately, not everything is perfect with the package and some bugs can be found such as this one. The function *dataset_imdb()* normally returns the dataset with some filter that we pass as parameters. However, the bug[2] found doing this tutorial is that when the parameter maxlen is less than 150 it returns a null for the test data.

```r
# Loads the dataset which is already split in train and test
# For some reason there is a bug. When adding the parameter Max len the function don't
# return test data if not above maxlen 180
imdb <- dataset_imdb(maxlen = maxlen, seed = 123 )
imdb_hack <- dataset_imdb(maxlen = 180 , seed = 123)
imdb$train
imdb_hack$test
# Seperate the labels and the data for train and test
c(train_data, train_labels) %<-% imdb$train
```

```
c(test_data, test_labels) %<-% imdb_hack$test

# This is the word index which can be interprate as a dictionnary betweend index to words
# (e.g.) the word "modestly" is represented by the index 20608
# by default this word to index contains 88584 words
max_features
# Convert this word to index List to a more friendly data frame
word_index_df <- data.frame(
  word = names(word_index),
  idx = unlist(word_index, use.names = FALSE),
  stringsAsFactors = FALSE
)

# The first 4 indices are reserved for 4 different specials keys
# PAD:    This is to fill a sequence that is shorter than Max lenght
# START:  This is to fill a sequence that is shorter than Max lenght
# PAD:    This is to fill a sequence that is shorter than Max lenght
# UNK:    This if for an unkown words that are not part of vocabulary
# UNUSED: This if for an unused word part of the vocabulary

word_index_df <- word_index_df %>% mutate(idx = idx + 3)
word_index_df <- word_index_df %>%
  add_row(word = "<PAD>", idx = 0)%>%
  add_row(word = "<START>", idx = 1)%>%
  add_row(word = "<UNK>", idx = 2)%>%
  add_row(word = "<UNUSED>", idx = 3)
# Sort the rows by index
word_index_df <- word_index_df %>% arrange(idx)
# A function that will decode a id an return the full review
decode_review <- function(text){
  paste(map(text, function(number) word_index_df %>%
              filter(idx == number) %>%
              select(word) %>%
              pull()),
        collapse = " ")
}
```

```
decode_review(train_data[[5]])
[1] "<START> excellent episode movie ala pulp fiction 7 days 7 suicides it doesnt get more
depressing than this movie rating 8 10 music rating 10 10"
```

The data cannot be used directly in the model since each sentence have different lengths. The *pad_sequences()* add elements in order to have a matrix with no holes.

```
# This function is built in the Keras environmet and as the name indicates it
# is use for padding sequences such as our reviews. after this is done all the
# review will have the same lenght. The shorter ones will be filled with the token
# <pad>. This applies to the train and test data.
train_pad <- pad_sequences(
  train_data,
  value = word_index_df %>% filter(word == "<PAD>") %>% select(idx) %>% pull(),
  padding = "post",
  maxlen = maxlen
)
```

```r
test_pad <- pad_sequences(
  test_data,
  value = word_index_df %>% filter(word == "<PAD>") %>% select(idx) %>% pull(),
  padding = "post",
  maxlen = maxlen
)
#Create a validation dataset
n = as.integer(round(nrow(train_pad)*0.7))
x_val <- train_pad[1:n, ]
partial_x_train <- train_pad[(n+1):nrow(train_pad), ]

y_val <- train_labels[1:n]
partial_y_train <- train_labels[(n+1):length(train_labels)]
```

Now with the data ready it is time to create the model and compile it.

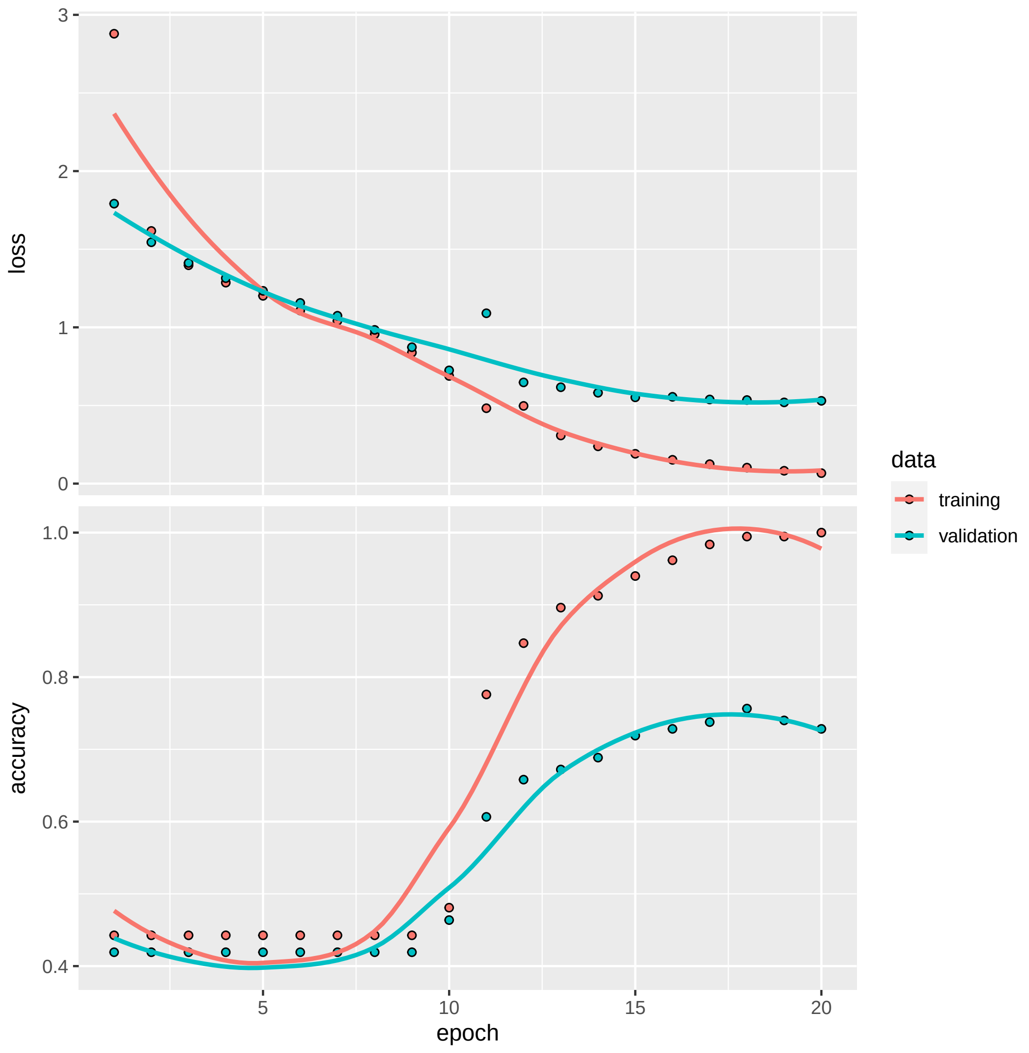
```r
model <- keras_model_sequential()
model %>%
  layer_embedding(input_dim = max_features, output_dim = 32) %>%
  layer_lstm(units = 24,
             input_shape = 32,
             batch_size = 8)%>%
  layer_dropout(rate = 0.1) %>%
  layer_dense(units = 1, activation = "relu")

model %>% compile(
  optimizer = 'adam',
  loss = 'binary_crossentropy',
  metrics = list('accuracy')
)
```

To train the model it is the same idea as the FNN and the CNN

```r
start_time <- Sys.time()
history_rnn_keras <- model %>% fit(
  partial_x_train,
  partial_y_train,
  epochs = 20,
  validation_data = list(x_val, y_val),
)
end_time <- Sys.time()
time_rnn_keras = end_time - start_time

results <- model %>% evaluate(test_pad, test_labels)
```

The training history can be plot by calling the history :

```r
plot(history_rnn_keras)

## `geom_smooth()` using formula 'y ~ x'
```



The accuracy of the model is as follow :

```
results
```

```
## $loss
## [1] 0.6962709
##
## $accuracy
## [1] 0.6293246
```

and the time it took to train is :

```
## Time difference of 20.01908 secs
```

5.3 h2o

5.3.1 Installation

```
install.packages("h2o")
library(h2o)
h2o.init()
```

By default, H2O Deep Learning uses an adaptive learning rate (ADADELTA) for its stochastic gradient descent optimization. **Prerequisites to launch H2o**, 64 bit Java 6+ if you want to open a h2o model that s more than 1 GB.

5.3.2 FNN

First of all we need to load the data and in order to do so we will use the datasets packages. We the data is in the workspace it will be divided into the train and test.

```r
library(datasets)
# Download the Iris data to workspace
data(iris)
# Change the data Iris output for string
iris$Species = sapply(as.character(iris$Species),
                      switch, "setosa" = 1,
                      "versicolor" = 2,
                      "virginica" = 3,
                      USE.NAMES = F)
# Here is another way, to split the data to
spec = c(train = .7,test = .3)
# Set a seed in order to be repreductible
set.seed(123)
# Sample through the dataframe using the sample and cut.
# The variable "g" returns a list of rows for train and
# test
g = sample(cut(seq(nrow(iris)),
               nrow(iris)*cumsum(c(0,spec)),
               labels = names(spec)
               )
           )
# Use the data and the row information to select rows
data_df = split(iris, g)
# Create vector that will contain X (Features variable)
# and Y the target variable
X = c()
Y = c()
X$train = as.matrix(data_df$train[,-ncol(data_df$train)])
X$test = as.matrix(data_df$test[,-ncol(data_df$test)])
Y$train = as.matrix(data_df$train[,ncol(data_df$train)])
Y$test = as.matrix(data_df$test[,ncol(data_df$test)],)
```

```r
# First we need to load the library
library(h2o)
h2o.init()
# Identify predictors and response

# We this package we need to convert everything in
# h2o format by casting.
h2o_df_train = as.h2o(data_df$train)
h2o_df_test = as.h2o(data_df$test)

# Labeling the target is optional but helpful to debug
y <- "Species"
x <- setdiff(names(h2o_df_train), y)
# For binary classification, response should be a factor
h2o_df_train[,y] <- as.factor(h2o_df_train[,y])
```

```
h2o_df_test[,y] <- as.factor(h2o_df_test[,y])

# The deeplearning is kept simple in order to give a high level of the user with
# minimal interaction
start_time <- Sys.time()
model_h2o <- h2o.deeplearning(x = x,
                              y = y,
                              training_frame = h2o_df_train,
                              epochs = 200,
                              seed=123)
end_time <- Sys.time()
time_fnn_h2o = end_time - start_time

summary(model_h2o)

perf <- h2o.performance(model_h2o, as.h2o(data_df$test))
pred <- h2o.predict(model_h2o, as.h2o(data_df$test))
1-mean(as.vector(pred$predict) != as.vector(data_df$test$Species))
```

The accuracy of this resources is

```
## [1] 0.9333333
```

and the time it took to train is :

```
## Time difference of 7.731083 secs
```

5.4 MXNET

5.4.1 Installation

For CPU :

```
install.packages("https://s3.ca-central-1.amazonaws.com/jeremiedb/share/mxnet/CPU/mxnet.zip",
                 repos = NULL)
```

5.4.2 FNN

First of all we need to load the data and in order to do so we will use the datasets packages. We the data is in the workspace it will be divided into train and test.

```
library(datasets)
library
data(iris)
set.seed(123)
iris$Species = as.numeric(iris$Species) - 1
spec = c(train = .7,test = .3)
g = sample(cut(seq(nrow(iris)),nrow(iris)*cumsum(c(0,spec)),labels = names(spec)))
data_df = split(iris, g)
X = c()
Y = c()
X$train = as.matrix(data_df$train[,-ncol(data_df$train)])
X$test = as.matrix(data_df$test[,-ncol(data_df$test)])
Y$train = data_df$train[,ncol(data_df$train)]
Y$test = data_df$test[,ncol(data_df$test)]
```

Then the model creation can start. As one can see, the encapsulation is also present with this package (The lego toy analogy).

```
library(mxnet)
data <- mx.symbol.Variable("data")
fc1 <- mx.symbol.FullyConnected(data, name="fc1", num_hidden=8)
act1 <- mx.symbol.Activation(fc1, name="relu1", act_type="relu")
fc2 <- mx.symbol.FullyConnected(act1, name="fc2", num_hidden=3)
softmax <- mx.symbol.SoftmaxOutput(fc2, name="sm")
mx.set.seed(123)

start_time <- Sys.time()
model_mx <- mx.model.FeedForward.create(
  softmax,
  X=as.array(X$train),
  y=as.numeric(Y$train),
  ctx=mx.cpu(),
  num.round=200,
  verbose = TRUE,
  eval.metric = mx.metric.accuracy,
  array.batch.size=5,
  learning.rate=0.07)
end_time <- Sys.time()
time_fnn_mxnet = end_time - start_time


preds = predict(model_mx,as.array(X$test),array.layout = "rowmajor")
summary(model_mx)
pred.label.mxnet <- max.col(t(preds)) - 1
1-mean(pred.label.mxnet != as.vector(data_df$test$Species))
```

The accuracy of this resources is

```
## [1] 0.9333333
```

and the time it took to train is :

```
## Time difference of 6.597001 secs
```

6 Discussion

6.1 FNN

The CRAN platform includes a lot of packages allowing to use different types of FNN, going from one hidden layer FNN to deep FNN. Vanilla FNN and deep FNN can be easily implemented and plotted with the package **neuralnet**, while more advanced architecture such as Bayesian regularized neural networks are available with the package **brnn**. If you decide to use one of the packages available on CRAN, we highly recommend to read the package documentation since default parameters, backpropagation algorithms and activation functions may vary from one package to another. However, for building deep FNN in R, we highly suggest to use Keras API, as it will give you much more flexibility and will allow you to use the latest deep FNN architectures. Keras has a lot of advantages such as building your model layer by layer (Lego toy analogy) and using different types of hidden activation functions, and different output activation functions such as softmax.The h2o package was a really nice discovery because of its simplicity. It offers a minimal interface but can easily be tuned with more than 100 parameters. The documentation and community is pretty healthy and an exploration of this environment is strongly suggested. Last but not least, the MXnet was a bit disappointing for many reasons. First of all, the documentation is not helpful in most of the case and it feels almost unnatural to use this package with R. Like we explain before, the package was mainly intended to be portable and cloud base for the industry deployment which could explain the lack of community.

6.2 CNN

The CNN models were simple and straightforward to implement in R with Keras API. Keras allows to build the CNN layer by layer, which is a great way of having control on its architecture. We saw that training and testing the model could be done in a few lines in R and that it is also possible to print the model directly within R console in order to check the number of trainable parameters. Keras also offers an automatic way of plotting the loss and accuracy score on the training and validation dataset while training the model, which gives a good idea of the evolution of your model performance per epoch. To sum up, Keras API offers a large number of tools to comfortably build any CNN model you would need to build in R.

6.3 RNN

The RNN was only tested on one package which sincerely was not bad nor good. Some bugs were discovered while using Keras, especially while downloading datasets. Therefore, it is recommended to carefully check the data at each step of the model construction. On the other hand, the data manipulation, the text embedding and the model training was simple to implement and visualize. Even if the data had to be padded in order to have a fix length everything that can be done in python felt pretty natural in R as well. The complexity of the package is the same. At the end of the day, if someone feels more comfortable in R stay assure that this tool can provide everything you need for RNN. The cran package rnn [15] had some promising architecture for LSTM ,GRU and/or simple vanilla RNN. However, the available examples were minimal and the package seems deprecated with the lacks of update on the GitHub repositories.

7 Conclusion

In the light of the discoveries we made while building this deep learning tutorial in R, we can conclude that deep learning in R is a new research area as this field was still emerging these last few years (figure 13). Even though R has been recognized as a major statistical software and has been used by data scientists around the world, building the latest and more advanced models can be limited compared to other programming languages such as Python. However, as these models will be built through APIs such as Keras that can be implemented on different programming languages, the skills that are developed in R while building these models, we still useful as they can always be leveraged.

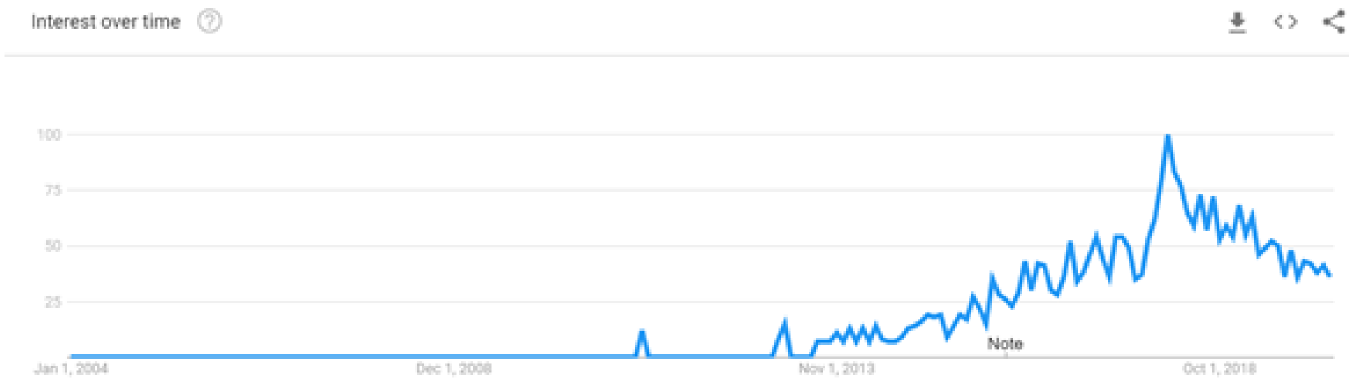


Figure 13: Subject: deep learning with R.

References

[1] What is the difference between ai, machine learning and deep learning? `https://www.geospatialworld.net/blogs/difference-between-ai%EF%BB%BF-machine-learning-and-deep-learning/`. Accessed: 2020-02-15.

[2] Test data not present when adding parameter maxlen. URL `https://github.com/rstudio/keras/issues/1007`. Accessed: 2020-03-28.

[3] Various ways to evaluate a machine learning model's performance. `https://towardsdatascience.com/various-ways-to-evaluate-a-machine-learning-models-performance-230449055f15`. Accessed: 2020-03-25.

[4] JJ Allaire Alfonso R. Reyes, Daniel Falbel. *rTorch: R Bindings to PyTorch*, 2019. URL `https://cran.r-project.org/web/packages/rTorch/rTorch.pdf`. R package version 0.0.3.

[5] JJ Allaire and François Chollet. *keras: R Interface to 'Keras'*, 2019. URL `https://cran.r-project.org/web/packages/keras/keras.pdf`. R package version 2.2.5.0.

[6] JJ Allaire and Daniel Falbel. *tensorflow: R Interface to 'TensorFlow'*, 2019. URL `https://cran.r-project.org/web/packages/tensorflow/tensorflow.pdf`. R package version 2.0.0.

[7] Marcus W. Beck. *NeuralNetTools: Visualization and Analysis Tools for Neural Networks*, 2018. URL `https://cran.r-project.org/web/packages/NeuralNetTools/NeuralNetTools.pdf`. R package version 1.5.2.

[8] Laurent Charlin. University Lecture. URL `http://www.cs.toronto.edu/~lcharlin/courses/80-629/slides_nn.pdf`.

[9] F. Chollet and J.J. Allaire. *Deep Learning with R*. Manning Publications, 2018. ISBN 9781617295546. URL `https://books.google.ca/books?id=xnIRtAEACAAJ`.

[10] Giuseppe Ciaburro and Balaji Venkateswaran. *Neural Networks with R: Smart models using CNN, RNN, deep learning, and artificial intelligence principles*. 2017.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL `http://www.deeplearningbook.org`.

[12] H2o.ai. *h2o: R Interface for the H2O Scalable Machine Learning Platform*, 2019. URL `https://cran.r-project.org/web/packages/h2o/h2o.pdf`. R package version 3.28.0.4.

[13] Christopher Olah. colah's blog, Aug 2015. URL `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

[14] Paulino Pérez-Rodríguez, Daniel Gianola, Kent Weigel, Guilherme Rosa, and Jose Crossa. Technical note: An r package for fitting bayesian regularized neural networks with applications in animal breeding. *Journal of animal science*, 05 2013. doi: 10.2527/jas.2012-6162.

[15] Bastiaan Quast and Dimitri Fichou. *Implementation of a Recurrent Neural Network architectures in native R*, 2019. URL `https://cran.r-project.org/web/packages/rnn/rnn.pdf`. R package version 0.9.8.

[16] Brian Ripley and William Venables. *nnet: Feed-Forward Neural Networks and Multinomial Log-Linear Models*, 2020. URL `https://cran.r-project.org/web/packages/nnet/nnet.pdf`. R package version 7.3-13.

[17] Paulino Perez Rodriguez and Daniel Gianola. *brnn: Bayesian Regularization for Feed-Forward Neural Networks*, 2020. URL `https://cran.r-project.org/web/packages/brnn/brnn.pdf`. R package version 0.8.

[18] Xiao Rong. *deepnet: Deep learning toolkit in R*, 2015. URL `https://cran.r-project.org/web/packages/deepnet/index.html`. R package version 0.2.

[19] Frauke Guenther Stefan Fritsch and Marvin N. Wright. *neuralnet: Training of Neural Networks*, 2019. URL `https://cran.r-project.org/web/packages/neuralnet/neuralnet.pdf`. R package version 1.44.2.

[20] Tong He Anirudh Acharya Tianqi Chen, Qiang Kou. *mxnet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems*, 2020. URL `https://mxnet.apache.org/api/r/docs/api/R-package/build/mxnet-r-reference-manual.pdf`. R package version 2.0.0.