Méthodes avancées en exploitation de donnée

(MATH80619)

Estefan Apablaza-Arancibia	11271806
Adrien Hernandez	11269225

March 28, 2020

1 Introduction

In the first section of this paper, a literature review covers the neural networks and deep learners algorithms, focusing on different type of neural networks architecture; the purpose of adding multiple hidden layers and, ultimately, what are the challenges regarding the increase in computing time. Furthermore, in the methodology section, a list of deep learning projects are shown in order to understand some patterns and methods. Then, an exhaustive list of the R libraries allowing to build neural networks and deep learners models. Correspondingly, the advantages and disadvantages of each libraries, 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 real data.

2 Litterature Review

2.1 What is Deep Learning and why use it?

Deep learning is a subset field of articial 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 extration 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 models 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

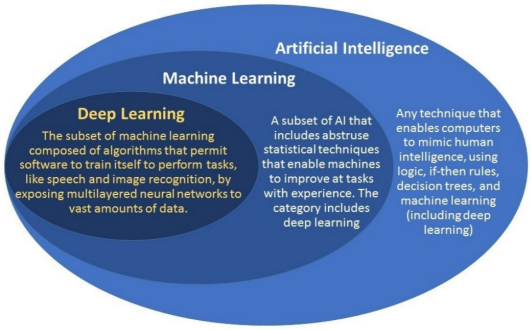


Figure 1: Artificial Intelligence vs Machine Learning vs Deep Learning

Since the last decade, deep learning models have shown notable predictive power and have been revolutionizing an important number of domain 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 ([**?** , Chapter 1]).

2.1.1 Feed Forward Neural Network

"Deep feedforward networks, also called feedforward neural networks, or multilayer perceptrons (MLPs), are the quintessential deep learning models." [**?**]

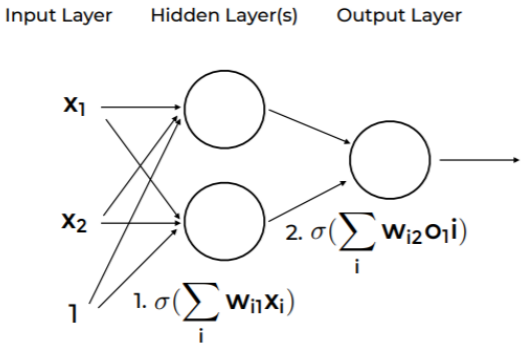


Figure 2: feedforward neural networks

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. **??**) 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 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:

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

Figure 3: Activation functions: output units. HEC

A detailed explanation of the theory of feedforward neural network can be found in [**?** , Chapter 6]

2.1.2 CNN

The CNN are a modified architecture of FNN that leverage the feature engineering that used to be hand maded by domain experts. This class of deep neural network are commonly use for image recognition and classification that can serve different applications such as facial recongnition, 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 perfom very well.

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 fonction to the convolutional layer output in order to increase the non linearity of the images before using 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 minimise a cost function, most of the time a categorical cross-entropy if we are performing a multi-class classification.

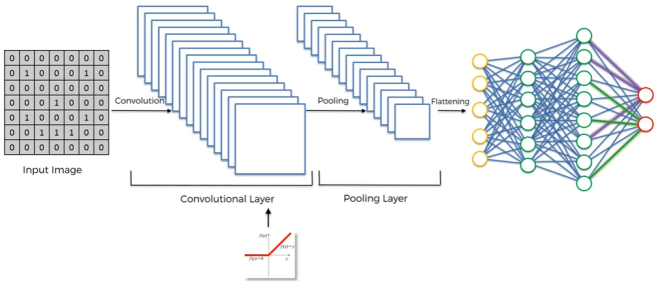


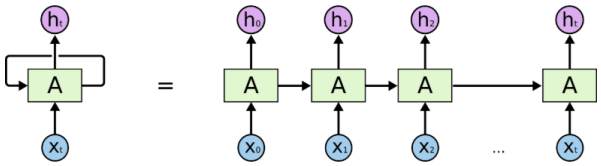
Figure 4: CNN, Deep Learning A-Z by SuperDataScience

To get a detailed explanation of convolutional neural networks we recommand to read [**?** , Chapter 9].

2.1.3 RNN

2.1.3. Add more info about this topics

Recurrent neural networks are a more advanced type of neural networks 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 sequantial data as well as what they have learned over time. One of their other advantage is to be able to handle input and ouput 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" Olah.



An unrolled recurrent neural network.

Figure 5: RNN, Understanding LSTM Networks by Olah

To get a detailed explanation of the theory of recurrent neural networks we recommand to read [**?** , Chapter 10]. We also recommand reading this blog post `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`

2.2 Deep Learning integration in R

The integration of Deep learning in R can be separate in two part ; (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 a already known software installation. This approach is not always trivial to install nor to manipulate but in 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 restraing 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 use 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 base on the same training and validation dataset and second indicator is about the time it took to execute the same task on differents ressources.

3.1 Workflow

3.1.0. Add more info about this topics

3.2 Dataset

3.2.1 FNN

For the FNN, the training data will be the well known *Iris* dataset. It contains four covariables and one categorical target variable.

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 color (Red-Green-Blue) that are 32 pixels by 32 pixels. The testing dataset has a total of 10 000 images

```
library(keras)
cifar10 = dataset_cifar10()
```

3.2.3 RNN

The dataset use to test an RNN is the IMDB Movie reviews sentiment classification. This pack

3.3 Characteristic

3.3.1 Accuracy

3.3.1. Add more info about this topics

Classification rate for.

3.3.2 Time elapsed

Not generalised but gives an idea about the training time.

4 Available resources

Like explain earlier this section will be divided in two in order to really differantiate the R packages standalone of the ressources that add an application interface.

4.1 R packages

4.1.0. à voir si on a encore de la place pour mettre un tableau récapitulatif

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.

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

```
neuralnet(formula,
      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
#' @param Y~X1+...+Xn
#' @param hidden = 1
#' @param threshold = 0.01
#' @param stepmax = 1e+05
#' @param rep = 1
#' @param algorithm = "rprop+"
#' @param err.fct = "sse"
#' @param act.fct = "logistic"
#' @param linear.output = TRUE
#' @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 more tricky 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 addind 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 custome activation functions.

Cons

- One of the disadvantage of this package is that it requires some data preprocessing as it only works with numeric inputs. Therefore, factor variables will need to be transformed into dummies during the preprocessing 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 number of iretation 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.

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 =
#' @param Y =
#' @param size =
#' @param linout =
#' @param entropy =
#' @param softmax =
#' @param maxit =
#' @param decay =
#' @return a nnet model
```

(b) Parameters

Figure 7: nnet package

Pros

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

Cons

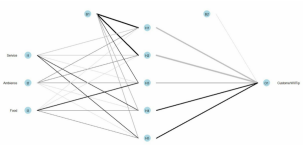
- It does not offer a lot of flexibility, and can only apply logistic sifmoid function for the hidden layer activation and cannot use tanH and ReLu.
- This package does not allow to use more that one hidden layer, and does not have any feature to find the optimal number of neurones in the hidden layer. It is up to the analyst to build a loop to test by cross-validation, for

exemple, the optimal hyperparameter values.

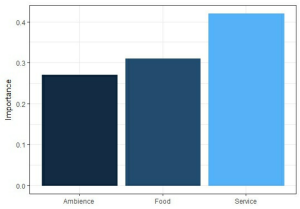
- Cannot use classical backpropagation algorithm to train the network. It only uses the BFGS (Broyden-Fletcher-Goldfarb-Shanno) which is a Quasi-Newton method and therefore increases the number of computation to reach a local optima. However, applied on a small dataset with a model having only one hidden 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 plooting, quantifying variable importance, conducting a sensitivity analysis, and obtaining a simple list of model weights."



(a) NeuralNetTools, plotnet function



(b) NeuralNetTools, Input variables importance

Figure 8: Embedding dimension hyper parameter

Pros

- Very easy to plot your neural network models with the function **plotnet(model)**.
- Visualize the input variables importance to the output prediction witht 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 specifcally 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".

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,
              cd = 1)
```

(a) Training function

```
#' @param X :
#' @param Y :
#' @param hidden :
#' @param activationfun :
#' @param learningrate :
#' @param momentum :
#' @param learningrate_scale :
#' @param output :
#' @param numepochs :
#' @param batchsize :
#' @param hidden_dropout :
#' @param visible_dropout :
#' @param cd :
#' @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 trough 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 linar, 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 inclusing both additive and dominance effects, and allows to takes advantage of multicore architectures via a parallel computing approach using openMP for the computations". (Pérez-Rodriguez, ...)

Important default parameters of the brnn function:

```r
brnn(X,Y,
      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,
      Monte_Carlo = FALSE
      ,tol = 1e-06,
      samples = 40)
```

(a) Training function

```r
#' @param X,
#' @param Y
#' @param neurons=2
#' @param normalize=TRUE
#' @param epochs=1000
#' @param mu=0.005
#' @param mu_dec=0.1
#' @param mu_inc=10
#' @param mu_max=1e10
#' @param min_grad=1e-10
#' @param change = 0.001
#' @param cores=1,
#' @param verbose=FALSE
#' @param Monte_Carlo = FALSE
#' @param tol = 1e-06
#' @param samples = 40
#' @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

Installation: You will need to install the diges package first through the function **install.packages(digest)** in order to download the rnn package.

Description Avaible from the CRAN platform, this package allows the "implementation of a Recurrent Neural Network architectures in native R, including Long-Short Term Memory (LSTM), Gated Recurrent Unit (GRU) and vanilla RNN." *Important default parameters of the rnn function:*

```
brnn(X,Y,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,Monte_Carlo = FALSE,tol = 1e-06, samples = 40,...)
```

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 changes 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 use 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 [**?**].

Pros

- Give a lot of flexibility since can basically be use to create all type of deep learning algorithms.
- Easy and fast prototyping
- Construct models exactly the same way as in python with a minimalist philosphy.
- 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.

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 enviromnent. 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."

Pros

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

Cons

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

4.2.3 MXNet

Description MXNet is a DL framework open source created by Apache. The framework is mostly backed by Intel, Microsoft and MIT. There is a manual[**?**] but itsn't part of the CRAN database.

Pros

- Allows the use of multiple CPU and multiple GPU, but it requires to download the package Rtools and a c++ compiler.
- Very easy to use and has a flexible implementation of diffenent 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.

4.2.4 rTorch

Description The rTorch CRAN package is to interface the popular open source machine learning library PyTorch[**?**]. The PyTorch was developed by Facebook AI Research lab (FAIR) and all the original features are available in rTorch.

Pros

- Large amount of functions to manipulate tensors.

Cons

- Doesn't have training capabilities since it is only for manipulations.

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 ot know the computation time remaining when running a model.

```
dl_fit3 <- h2o.deeplearning(x = x,
                            y = y,
                            training_frame = train,
                            model_id = "dl_fit3",
                            epochs = 50,
                            hidden = c(20,20),
                            nfolds = 3,                            #used for early stopping
                            score_interval = 1,                    #used for early stopping
                            stopping_rounds = 5,                   #used for early stopping
                            stopping_metric = "misclassification", #used for early stopping
                            stopping_tolerance = 1e-3,             #used for early stopping
                            seed = 1)
```

Figure 11: 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 amount of parameters, however, it doesn't give all the capability of other resources.

5 Examples

In order to give concrete deep learning examples, four resources were selected for tutorials; Neuralnet, Keras using Tensorflow backend, H2O, rTorch and MXNET.

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)

# 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.

```
# 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,]
```

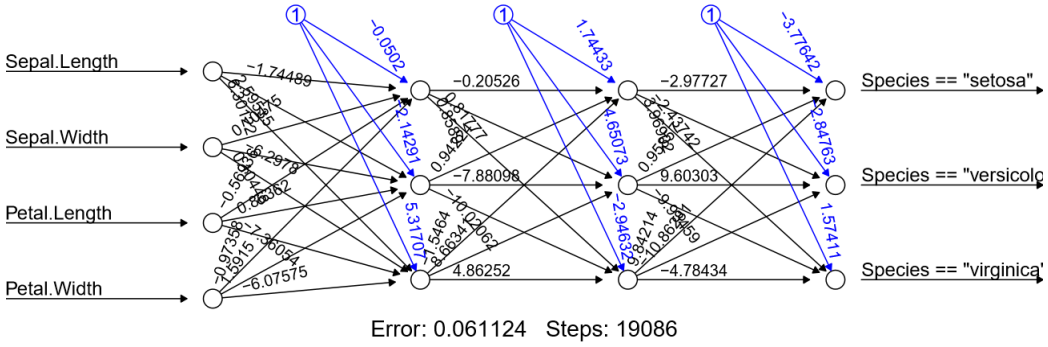
Then, we will build a deep FNN with 2 hidden layers of 3 neurons. As hyperparameters, we will use only 1 epoch, we will use the standard backpropagation algorithm and therefore the learning rate has to be specified. We will use a logistic activation function.

```
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:

```
plot(neuralnet_model)
```

```
## Error in plot(neuralnet_model): object 'neuralnet_model' not found
```



The model visualization allows to see if our model corresponds to the hyperparameters we used during training. However, this is not opitimized 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)
```

```
## Error in predict(neuralnet_model, iris_test): object 'neuralnet_model' not found
```

```
table(iris_test$Species, apply(neuralnet_prediction, 1, which.max))
```

```
## Error in table(iris_test$Species, apply(neuralnet_prediction, 1, which.max)): object 'iris_test'
not found
```

The accuracy of this resources is

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

and the time it took to train is :

```
## Error in eval(expr, envir, enclos): object 'time_fnn_neuralnet' not found
```

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")
```

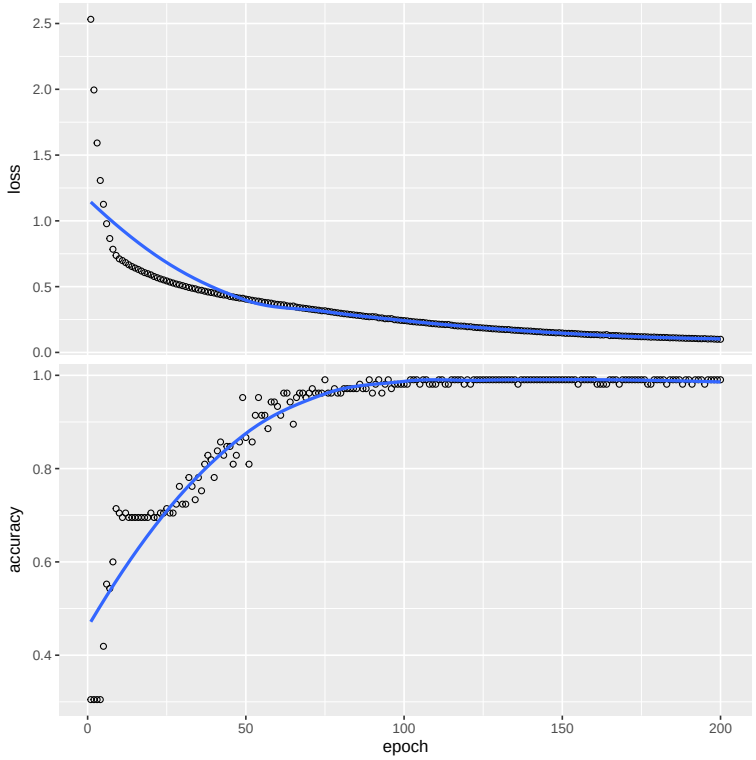
5.2.2 FNN

```
library(datasets)
library
data(iris)
set.seed(123)
iris$Species = sapply(as.character(iris$Species),
                      switch, "setosa" = 1, "versicolor" = 2, "virginica" = 3, USE.NAMES = F)
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$train = as.matrix(data_df$train[,ncol(data_df$train)])
Y$test = as.matrix(data_df$test[,ncol(data_df$test)],)
```

```
library(keras)


# create model
model_keras = keras_model_sequential()
model_keras %>%
  layer_dense(units = 8, activation = 'relu', input_shape = c(4)) %>%
  layer_dense(units = 3, activation = 'softmax')
# Compile model
model_keras %>% compile(
  loss = 'categorical_crossentropy',
  optimizer = optimizer_adam(),
  metrics = c('accuracy')
)
start_time <- Sys.time()
history <- model_keras %>% fit(
  as.matrix(X$train),to_categorical(Y$train),
   epochs=200, batch_size=5)
end_time <- Sys.time()

#Benchmark
time_fnn_keras = end_time - start_time
model_keras %>% evaluate(X$test, to_categorical(Y$test))
model_keras %>% predict(X$test, to_categorical(Y$test))

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))
```

This package of the possibility to store the training history :

```
plot(history)
```



The accuracy of this resources is

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

and the time it took to train is :

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

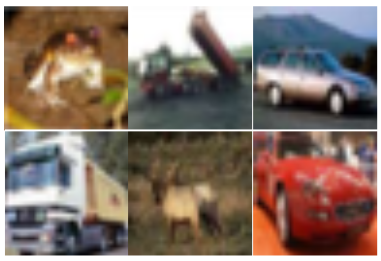
5.2.3 CNN

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

```
library(keras)
cifar10 = 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 use train our model with 64 images per iteration. We set epoch to 10, our model will go trough the entire training set 10 times. 20% of the training set will be used as validation set, in order to tune the parameters without biais.

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


batch_size <- 64
epochs <- 10
validation <- 0.2
```

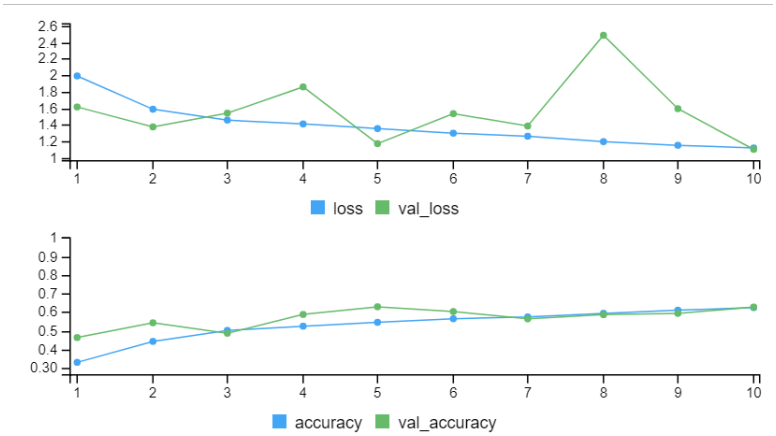
We will us as loss function the categorical_crossentropy that works very well with our softmax activaction function in order to peform multi label classification. We will look at the accuracy as the final metric to check the performance of our model on the test dataset.

```
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**.

```
history <- model %>% fit(
    X_train, Y_train,
    batch_size = batch_size,
    epochs = epochs,
    validation_split = validation,
    shuffle = TRUE)
```

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] 1.091844


$accuracy
[1] 0.6255
```

We can see that our accuracy is 62.55%.

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

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

5.2.4 RNN

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

```
library(h2o)
h2o.init()
# Identify predictors and response



h2o_df_train = as.h2o(data_df$train)
h2o_df_test = as.h2o(data_df$test)

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])

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
```

5.3.3 CNN

5.3.4 RNN

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)
library(mxnet)
```

5.4.2 FNN

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

6.2 CNN

6.3 RNN

7 RNN

References

[] 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.

[] 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.

[] 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.

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

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

[] 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.