First steps on deep learning for NLP by R's h2o package

Iñaki Inza, Borja Calvo

First steps on deep learning for NLP by R'2 h2o package

Deep learning [1] has emerged with disruptive energy in the machine learning community. As an inheritor of the well-known and pioneering neural networks (popular models for clustering and supervised classification specially during 90s) the deep learning paradigm has taken the best of them to solve high-dimensional, challenging modelization tasks: speech recognition, computer vision and why not, also your NLP domain.

For the sake of simplicity I will limit my discourse to the supervised classification scenario. A neural networks is a computational model inspired by the way biological neural networks in the human brain process information (see for a quick introduction to neural networks). A neural network is structured by an input (formed by the original variables-predictors of our problem) and an output layer (formed by the values of our class variable to be predicted), together with a set or intermediate-hidden layers which will try to map the inputs to the desired outputs by means of a training process.

The basic computation unit in all the intermediate-hidden layers is a neuron (also called node or unit), which receives as input a set of nodes of the previous layer. Hidden layers have a tunable number of neurons. The inputs are connected to the neuron with an associated weight, which is assigned on the basis of its relative importance to other inputs. The computation of these weigths is the core of the training process in neural networks.

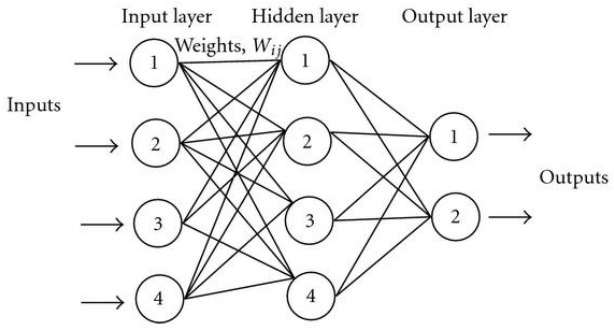


Figure 1: A neural network structure with one hidden layer

The neuron applies a so called activation function to the weighted sum of its inputs. Popular activation functions (sigmoid, tangent, etc.) are non-linear in order to model non-linear decision boundaries which discriminate the problem classes.

Año: EMLCT Master
Profesor(a): Iñaki Inza, Borja Calvo
Teléfono: 943 01 50 26
E-mail: inaki.inza@ehu.es

GIPUZKOAKO CAMPUSA
CAMPUS DE GUIPUZCOA
Paseo Manuel de Lardizabal, 1
20018 Donostia (Gipuzkoa)

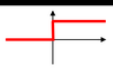
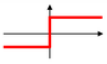
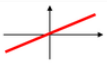
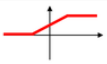
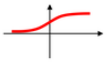
Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer NN	

Figure 2: A subset of popular activation functions

The most popular algorithm for training the weights of the neural network structure is called 'backpropagation. While all initial weights are randomly assigned, for every sample in the training dataset (known predictor values; known class value) a 'propagation' process is started: the inputs are fed to the input layer, activation functions calculated in hidden layers, and predicted output calculated. This predicted output is compared with the real class-value of the fed training sample, and the error is 'propagated back' to the previous layers. These backpropagated errors are used to adjust the weights in the network, flowing back in the network. An error signal is calculated in each neuron, commonly by a weighted summation of forward connected errors in the network (e.g. $\delta_2 = w_{24}\delta_4 + w_{25}\delta_5$). An updating of the weights is performed by calculating in each neuron the 'gradients'-derivatives of the activation and error functions with respect to the weights (follow this link for a graphical explanation of the backpropagation procedure). This is a non-trivial process which demands the use of computational-cost, optimization techniques such as 'Gradient Descent'. This process is repeated with all the training samples. Its objective is to reduce the error of the output layer in a next iteration-epoch, when training samples are fed again to the network with the updated weights. Several convergence criterias are used to stop the training process: a fixed number of epoch-iterations, stability of the weights, etc. The following link is a nice explanation about the learning and backpropagation procedures.

And... what is new in this business? Why is now deep learning 'the topic'? As I noted you before, the training of the network weights and specially its backpropagation stage, is a computational-cost process. The addition of more layers and neurons per layer is assumed to be able to model more complex, non-linear decision boundaries among problem classes-outputs. The availability of modern computational resources (e.g. parallel computing) and the recent proposal of more computationally-efficient training algorithms [1] is allowing the inclusion of a larger number of hidden layers and neurons (this is the origin of the 'deep' term) which are able to model problems of large input-dimensionality.

As a challenging domain of high-dimensionality, NLP has raised the attention of the deep learning community to apply their last methodological developments. The NLP community is also including this methodology in its portfolio of algorithms [2]: this reference is a popular tutoral on deep learning for NLP, and different versions can be found in the associated webpage.

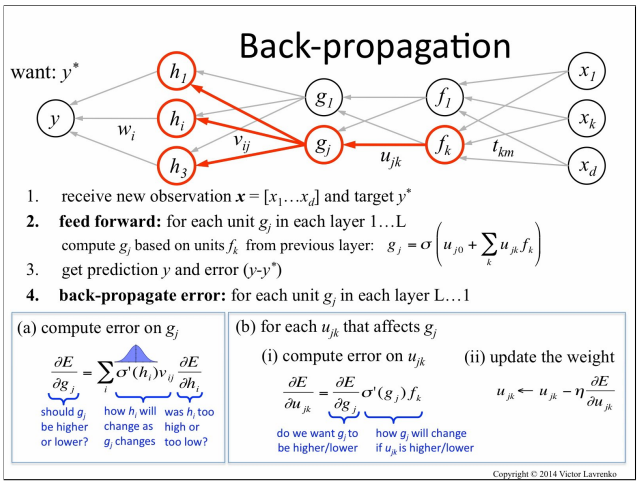


Figure 3: Ilustration of the bakcpropagation process of the error. Feedforward of training samples from rigth to left. Backpropagation from left to right.

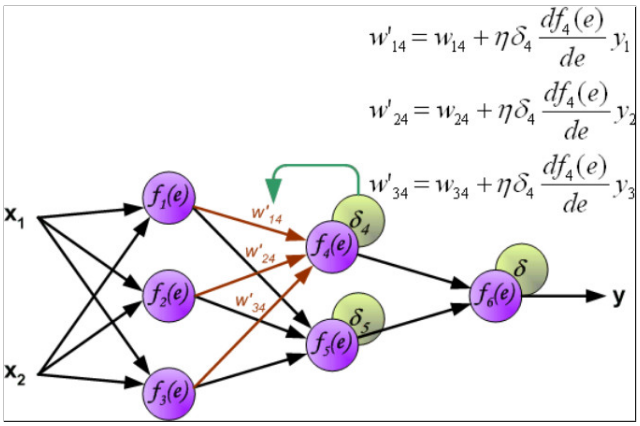


Figure 4: Updating of weights in neuron 4 after calculating its associated backpropagated δ_4 error. Backpropagation from rigth to left.

> Several sectors of the machine learning community criticize the neural network paradigm because of the 'black box' nature, non-transparency and low-level of comprehensibility of the learned model. In contrast to other machine learning algorithms whose final model can be understood by final users and experts (e.g. decision trees, Bayesian networks), the complex structure of neurons, connections and weights is the core of the neural network approach. Discussions about the this issue are plentiful.

However, different research groups have shown very competitive results in challenging domains. These results, compared to other machine learning algorihtms, have been specially remarkable in domains composed of a large number of predictors-variables with a low correlation degree with respect to the class to be predicted ('low-level features'). For example, in computer vision, where common datasets are composed of thousands of pixels: how does a single pixel help in predicting the object of an image, which is its correlation degree with respect to the image type? For example, in NLP domains, where common datasets are composed of thousands of unigrams: how does a single unigram help in predicting the document type, which is its correlation degree with respect to the document type? Both our intution and analytical results say that, extremely low.

Starting from these near-non-informative, low-level predictors, a single layer neural network won't hardly be able to model the problem output's non-linearity. Thus, the 'deep learning miracle' supposedly resides in the concatenation of many layers (and neurons per layer) which will form 'more advanced abstraction levels' and will accurately model the decision boundary between problem classes. The deep learning paradigm supposedly avoids in the initial step to the dataset designer the responsability in the engineering-design of more advanced features with a larger correlation degree with the output. Further layers in the deep structure should be able to substitute these informative features and should be able to complement among them in order to finally model the problem output. The price to be paid is a complex and difficult-to-understand model, together with a high computational cost.

A set of attractive deep learning software tools and platforms have been proposed during the last years: Keras, Caffe, Google's TensorFlow, Microsoft's CNTK... Due to the large computational demands of a deep learning procedure, needed properties are an efficient storage of the raw dataset and parallel programming. While R's `caret` package offers access to the `deepnet` package, this does not perform parallel computation. Thus, the `h2o` package is an attractive alternative for our purposes. In spite of the computing environment of the user, it offers efficient storage and parallel computation: using all the computing core-units of your laptop,

or accesing a complex Hadoop-computing-cluster architecture. `h2o` package programmers have coded many `R` basic routines and a set of popular clustering and classification algorithms in parallel form, offering efficient storage. Among them, a deep learning implementation is offered.

> `h2o` incorporates networks of the popular *multi-layer-feed-forward* type. You will discover that there are other types of deep networks such as *convolutional neural networks (CNN)* for image and text processing, *long-short-term-memory (LSTM)* for time series, *autoencoder* models for outlier detection and one-class-classification, etc..

The following lines will only be a humble and short tutorial to the use of the deep learning capabilities of `h2o`. This package offers a large set of routines for data manipulation, preprocessing, dimensionality reduction, clustering, supervised learning, etc. However, we will focus on the those needed for a supervised deep learning model computation.

First, it is needed to 'activate a computing cluster' and activate a connection. Thus, two key factors of this initializations are the memory size and the number of computing CPU cores. When using our own laptop for computation, we can use all its computing cores by setting `nthreads=-1`. In case of having access to for example, a computing cluster based on Hadoop technology, the initialization only needs an IP address and port number.

```
library(h2o)
# set the h2o connection to the computing unit
h2o.init(ip='localhost', port=54321, nthreads=-1, max_mem_size = '4g')
# information about the computing unit
h2o.clusterInfo()
```

First, we will make use of the `IMDB.csv` file of our working directory. It is composed of 1000 positive and 1000 negative film reviews. Comments have been preprocessed with `WEKA's StringToWordVector` and `NumericToBinary` filters. The file is saved in an `h2o's hex` type internal object which provides an efficient storage designed for big data configurations.

```
setwd("/home/inza/Escritorio")
imdb.hex = h2o.importFile(path="/home/inza/Escritorio/IMDB.csv", destination_frame="imdb.hex")
dim(imdb.hex)
h2o.table(imdb.hex$C1)
h2o.table(imdb.hex$C2)
# or even the file can be uploaded from its WEKA format, thus saving original column names:
# imdb.hex = h2o.uploadFile(path="/home/inza/Escritorio/IMDB.arff",
# destination_frame="imdb.hex", parse_type="ARFF")
# colnames(imdb.hex)
# consult loading options in this link:
# https://www.rdocumentation.org/packages/h2o/versions/3.10.2.2/topics/h2o.importFile
```

The labeled dataset is split-partitioned in training (66%) and testing (34%) partitions for evaluation purposes. You can already note that `h2o` forces the user to make use of specific functions for data manipulation, different to those of basic `R`.

```
imdb.split=h2o.splitFrame(data=imdb.hex, ratios=0.66)
imdb.train=imdb.split[[1]]
imdb.test=imdb.split[[2]]
dim(imdb.train)
```

It seems that everything is ready for learning a deep neural model. Sure. However, take a look to the help of the `h2o.deeplearning()` function: even in the `R` console or in this link. Impressive, no? Number of layers and neurons per layer, type of activation function, number of epochs, learning rate... How can this huge set of parameters be tuned for our problem? Open problems for the enthusiastic deep learning community. And your interest on the topic will be the one to go deeper in this huge set of parameters. It is true that this huge set of parameters to be tuned is one of the main deep learning criticisms. `h2o` offers this tutorial on its deep learning capabilities.

> Other resources and packages for deep learning in `R` can be found in the following link

One of the key parameters is `hidden`, fixing the number of layers and neurons per layer. As exposed, a larger number of layers and neurons per layer should learn more complex feature abstractions and decision boundaries: and this will test the computing capacities of your platform... After learning a model on the training partitions, it is used to predict the class on the test partition.

```
# fixing the predictors and output variable
# fixing three layers of 100 neurons each
modelDL <- h2o.deeplearning(x=2:1166, y=1, training_frame = imdb.train, hidden = c(100,100,100))
predictionsDL=h2o.predict(object=modelDL,newdata=imdb.test)
# the object shows the a-posteriori probabilities of belonging to each class for each test sample
predictionsDL
# calculate model performance. The output, first a confusion matrix and then a
# large set of performance statistics: the maximum value of each metric when
# changing the belonging probability threshold to the positive class (0.5, by default)
performanceDL=h2o.performance(modelDL,imdb.test)
performanceDL
# area under the ROC curve: true positives versus false positives
h2o.auc(performanceDL)
```

Word2vec and model learning

You know better than me that a **Word2vec** algorithm takes a text corpus as input and produces a vectorial representation of words as output. These vectors are numeric: they can be intepreted within an Euclidean space. It is needed to create a vocabulary from the train text data, and then learn the vector representation for the words. Instead of a `DocumentTermMatrix` representation, this vector representation of the words can then be used to learn machine learning models. `h2o` offers an interesting tool to learn word2vec representations. Let's do a short demo with a realistic NLP-BigData dataset consisting of 568,454 food reviews left by Amazon users. Data was used for a popular kaggle competition.

```
# Load the dataset from the web
# Composed of 10,000 reviews, 10 variables each.
# Text is in the last column. Score of the review, 4th
reviews = h2o.importFile("https://s3-us-west-2.amazonaws.com/h2o-tutorials/
                         data/topics/nlp/amazon_reviews/AmazonReviews.csv")
dim(reviews)
colnames(reviews)
h2o.table(reviews$Score)

# Create and append a new target column, considering as `positive reviews' EQG than 4
reviews$PositiveReview <- h2o.ifelse(reviews$Score >= 4, "1", "0")
h2o.table(reviews$PositiveReview)
```

```
# Tokenize words
# Set stop words
STOP_WORDS <- read.csv("https://s3-us-west-2.amazonaws.com/h2o-tutorials/data/topics/nlp/
                       amazon_reviews/stopwords.csv", stringsAsFactors = FALSE)$STOP_WORD
STOP_WORDS

# Create a tokenize function that tokenizes the sentences and filters certain words
tokenize <- function(sentences, stop_word = STOP_WORDS){
  # Tokenize sentences by word
  tokenized <- h2o.tokenize(sentences, "\\\\W+")
  # Convert words to lowercase
  tokenized_lower <- h2o.tolower(tokenized)
  # Remove words with one letter
  tokenized_filter <- tokenized_lower[(h2o.nchar(tokenized_lower) >= 2) | is.na(tokenized_lower), ]
  # Remove words with any numbers
  tokenized_words <- tokenized_filter[h2o.grep(pattern = "[0-9]", x = tokenized_filter,
                                               invert = TRUE, output.logical = TRUE), ]
  # Remove stop words
  tokenized_words <- tokenized_words[is.na(tokenized_words) | !(tokenized_words %in% STOP_WORDS), ]

  return(tokenized_words)
}

# Run the previous tokenization function from the reviews' text
words <- tokenize(reviews$Text)
head(words)

# Train a Word2Vec model of 10 dimensions
# It is a computationally demanding process with respect to the vector size
w2v_model <- h2o.word2vec(training_frame = words, vec_size = 10)
# Play with the learned embedding
h2o.findSynonyms(w2v_model, "coffee", count = 5)

# Trasform original words to vectors by means of the learnt Word2Vec model
review_vecs <- h2o.transform(w2v_model, words, aggregate_method = "AVERAGE")
head(review_vecs)

# Calculate indexes for a train-test partition, using the 50th quantile of Time variable
time_split <- h2o.quantile(reviews$Time, prob = 0.5)
reviews$Train <- h2o.ifelse(reviews$Time < time_split, "Yes", "No")
train <- reviews[reviews$Train == "Yes", ]
test <- reviews[reviews$Train == "No", ]
# Add aggregated word embeddings. Append columns
ext_reviews <- h2o.cbind(reviews, review_vecs)
# Split data into training and testing
ext_train <- ext_reviews[ext_reviews$Train == "Yes", ]
ext_test <- ext_reviews[ext_reviews$Train == "No", ]

# Train a deep learning predictive model
# Fix the predictors: 4 original and transformed vector's 10 dimensions
predictors <- c('ProductId', 'UserId', 'HelpfulnessNumerator', 'HelpfulnessDenominator',
```

```
                'Time', colnames(review_vecs))
response <- 'PositiveReview'
# Learn the model: any operation will take time. Be careful with parameters' values
# An idea to limit the computation: reduce the number of training samples with previous time_split
# Be patient: once the model is learnt, future usage for prediction is justified
modelDL <- h2o.deeplearning(x=predictors, y=response, training_frame = ext_train,
                            validation_frame = ext_test, hidden = c(10,10,10))

# Confusion matrix is computed on the test partition, `ext_test'
h2o.confusionMatrix(modelDL, valid = TRUE)
# Calculate variables' univariate relevance in the model
# I am not sure how it is calculated
h2o.varimp(modelDL)
h2o.varimp_plot(modelDL,5)
```

Proposed exercise

Selecting and formatting an NLP dataset of your interest with a large set of low-level features (unigrams?) or applying a word2vec procedure and building your own deep learning models is an attractive exercise. I hope you have ideas about where start looking for a dataset of your interest. Some of the datasets offered in our datasets' directory are adequate: read their descriptions.

Remember that WEKA's `StringToWordVector` filter can quickly help you extracting the unigrams from different texts and then exporting the resulted file to a csv format that can be load by `h2o`: the `text2matrix.R` function offered in our datasets' working directory offers a similar functionality. Remember also the preprocessing facilities offered by the `tm` package.

Comparing `h2o`'s deep learning results with other supervised techniques offered by the package is another part of the exercise; or comparing models learnt with or without word embedding (Word2Vec). While you can consult the list of supervised algorithms offered by `h2o`, it shows popular algorithms like naiveBayes, random forests or generalized linear models. Follow this link with simple advices about when to use deep learning or other popular machine learning algorithms. This exercise will also force you in the complex attempt of tuning the parameters of `h2o`'s deep learning.

References

[1] J. Schmidhuber. Deep learning in neural networks: an overview. *Neural Networks*, 61:85–117, 2015.

[2] R. Socher and C.D. Manning Y. Bengio. Deep learning for nlp (without magic). Technical report, Tutorial Abstracts of Association for Computational Linguistics 2012, 2012. http://www.socher.org/index.php/DeepLearningTutorial/DeepLearningTutorial.