이화여자대학교 대학원
2019학년도
석사학위 청구논문

A Study on Various Deep Learning Algorithms with R

통 계 학 과

박 정 민

2020

A Study on Various Deep Learning Algorithms with R

이 논문을 석사학위 논문으로 제출함

2019 년 12 월

이화여자대학교 대학원

통 계 학 과 Jeongmin Park

Jeongmin Park의 석사학위 논문을 인준함

지도교수 ____이은경____ ____

심사위원 ____차지환____ ____

____이은경____ ____

____이동환____ ____

이화여자대학교 대학원

Table of contents

List of tables

List of figures

Summary

In recent days, deep learning is getting more popular in almost every field, such as bioinformatics, medical images, finance, as well as computer science, and so on. Most software for the deep learning algorithms are written in Python, mainly because of Tensorflow in backend tensor calculation. Nowadays, the realization of deep learning is also possible in R, which is more popular in the statistics area. This paper provides a guideline for the users for deep learning with R among various R packages for deep learning.

First, we briefly introduce various algorithms of deep neural networks and review state-of-art software for deep learning with R - deepnet, NeuralNetTools, automl, autoencoder, keras, RcppDL, mxnet, h2o. We also provide sample codes to help users better understand each function, and compare the performance of various R functions for deep neural networks with Portuguese bank marketing data.

I. Introduction

Artificial Intelligence (AI), machine learning (ML), and deep learning (DL) are all terms that can be easily heard from literally everywhere nowadays. Among these terms, deep learning is thought to be the core technology to lead the Fourth Industrial Revolution and in fact, it is already in every corner of one's life. For instance, smartphone speech recognition service, object recognition service, self-driving car, phrase-based machine translation are all innovative technologies based on deep learning algorithms. These techniques are also applied to various fields such as bioinformatics, the medical industry, finance, computer science and so on.

As the popularity and usage for these methods are getting higher, there is a growing interest in how to implement these algorithms. In line with the growing demand for deep learning, there are numerous amounts of contents that provide information about the use of deep learning. Most of the existing manuals are written in Python as it runs fast and is easy to learn. However, in comparison to the overflowing guidelines for Python, the users who use R are experiencing discomfort due to the lack of information.

The main purpose of this paper is to provide guidelines for users who use deep learning in R. In Section 2, we briefly cover various deep learning algorithms such as DNN, CNN, RNN, LSTM, GRU, AE, and so on. In Section 3, we introduce R packages and functions that are used for building deep learning structures. The packages handled in this thesis are categorized into three groups according to what software they are based on; R-based, Python-based, and other. This part also includes sample codes to help users better understand each function intuitively. In Section 4, we use a Portuguese bank marketing data to compare the accuracy and computation time of each R function. In the last section, Section 5, we summarize the overall results and finish the thesis.

II. Review

Deep learning is a concept based on artificial neural networks with multiple layers. The main algorithm of deep learning works by extracting features in each layer that represents the data well, using non-linear modules. As it reaches deeper into the layer, each layer is capable of detecting more complex and specific features. The key characteristic of deep learning is that they teach themselves to find representations needed for detection. Deep learning models can have various structures such as deep neural networks, convolutional neural networks, recurrent neural networks, and so on.

The following content briefly covers the mechanisms of each deep learning architecture and R packages that can be used to implement each structure.

A. Deep Neural Network

Deep Neural Networks (DNN) are usually defined as networks that have an input layer, output layer and at least 2 hidden layers. Each of these layers are made up of a collection of neurons. The algorithm works by neurons receiving as input the neuron activations from the previous layer and then performing a simple computation. Then it passes the outcome to the following layer until it reaches the output layer. The neurons of the network jointly implement a complex nonlinear mapping in each step. DNNs use fully-connected-layers which results in a large amount of computation. The optimal mapping is learned from the data by adjusting the weight of each neuron using a method called backpropagation.

B. Convolution Neural Network

Convolution Neural Networks (CNN) is a class of DNN which adjusts some of the drawbacks of DNN. The architecture of this network is formed by a convolution layer, pooling layer, fully connected layer and an activation function. CNNs are often used in analyzing visual imagery like image recognition and video analysis.

The convolution layer convolves the input layer with a multiplication or dot product and passes the result to the following layer. In this process, the network uses filters that share the same weights, which significantly reduces the number of parameters.

The pooling layer partitions the input into a set of non-overlapping rectangles stressing the importance of its rough location. In this step many different types of pooling units can be used, such as max pooling, average pooling, and so on. This layer takes part not only in reducing the number of parameters but also in controlling overfitting.

After several convolution and pooling layers, the neural network finally reaches the fully connected layer. The flattened matrix from the previous step goes through the fully connected layers. Fully connected layers connect every neuron in all the layers.

The activation function is implemented after the convolution layer and defines the output of each node. It usually acts as an abstraction representing the rate of action potential firing in the cell. Depending on the type of data the network is handling, different types of activation structures can be used. Relu, sigmoid function, hyperbolic tangent is some of the most frequently used activation functions.

Other than the structure of CNN, improvements of the model can be done by choosing the optimal hyperparameters such as the number of filters, the shape of the filters, dropout rates and so on.

C. Recurrent Neural Network

Recurrent Neural Network (RNN) is a class of artificial neural networks also known as a feed forward neural network. The main characteristic of RNN is that it is a sequence model with the ability to ‘remember’. This process is done in the hidden layer, also known as a ‘hidden state’. Thanks to this feature, RNN can handle various lengths of input, including even very long sequences. The biggest strength of RNN is that it can have a very flexible structure. As a result, RNNs are capable of handling tasks such as handwriting recognition, speech recognition, and sentiment analysis.

RNN is made up of an input layer, output layer and a hidden layer. Every time the information is put into the network sequentially, the network updates its memory according to the previous hidden state. This part is called feed-forward propagation. The model then tunes its parameters until the loss of the model is minimized, and this part is called back propagation. The parameters are the same for each state and this process is repeated until the end of the input. This is how RNN got the name ‘recurrent’.

D. Long Short-Term Memory

Long Short-Term Memory (LSTM) is an architecture designed to overcome a weakness in RNN. RNN is a powerful structure of neural networks as it can deal with a long sequence of input. However, due to this uniqueness, vanishing gradient problem occurs, which results in a poor learning ability when the related information is far away from one another.

LSTM overcame this drawback in RNN by adding a ‘cell-state’ in the hidden state. A cell-state works like a conveyor belt and is controlled by a tool called a ‘gate’. There are three gates in LSTM. The first is the forget gate layer which decides whether to keep past information or not. The second gate is the input gate layer which decides the amount of new information to store. Then a new cell-state value is created. Next, using the values in the forget gate, input gate, previous cell-state, newly created cell-state, an updated cell-state is generated. Lastly, the third gate is the output gate layer. The value of output is determined in the output gate.

RNN and LSTM are both widely used and either architecture can have a better performance than one another depending on the data.

E. Gated Recurrent Units

Gated Recurrent Units (GRU) is a deep neural network architecture derived from RNN and LSTM. GRU suggests a solution to the relatively long computation time of LSTM, with a simpler structure leading to a smaller amount of computation.

The structure of GRU is similar to LSTM but there is a distinctive difference in the number of gates. Unlike LSTM which has three gates, GRU has only two gates. These gates are the reset gate and the update gate. Both of the gates are set to have a value between 0 and 1 in order to perform convex combinations. The reset gate controls how much of the previous state it wants to remember, capturing the short-term dependencies of the time series. Thus, the more the value is closer to 1, it is interpreted that the algorithm recovers the previous state. On the contrary, when the value is close to 0, the previous hidden state is reset to defaults. The update gate functions analogous to the forget gate and input gate in the LSTM structure. It controls how much of the new state reflects the old state, capturing the long-term dependencies of the time series. This gate decides the amount of past information that needs to be passed along to the future.

Even though GRU has a relatively simplified structure and is receiving a lot of attention, it is hard to say that it is a better architecture than RNN or LSTM. This is due to the fact that the optimal structure is different depending on the type of dataset.

F. Auto-Encoder

An Auto-Encoder (AE) is also an architecture based on artificial neural networks. It is consisted of an input layer, hidden layer, and an output layer. The main difference of AE from other deep learning architectures is that the input and output layer have the same number of nodes.

AE is constituted by two parts which are the encoder and decoder. The encoder maps the input data into a code and tries to find the distinctive features that represent the given data well. Then the decoder maps the code into its original form, which is called the reconstruction process. The main purpose of AE is to reduce the dimensionality, compress the data, and find an efficient way to express the given data. The auto encoder tries to find the most important features so that the network is capable of reconstructing the original data.

There are many variations of AE such as stacked auto encoder (SAE), denoising auto encoder (DAE), sparse auto encoder (SAE), variational auto encoder (VAE). Even though they have slightly different structures, the main purpose of the architecture is all the same. AEs are used in various fields like dimensionality reduction, PCA, image processing and so forth.

G. Restricted Boltzmann Machine

The Restricted Boltzmann Machine (RBM) is actually a shallow two layer net that only has one visible layer and one hidden layer. The name 'restricted' refers to the fact that no two nodes in the same layer share a connection. The main goal of RBM is to recreate the input as accurately as possible. Hence, RBM is a feature extractor neural network which belongs to the family of auto-encoders.

As mentioned above, the structure of RBM is very simple. In the forward training process, the inputs are modified by different weights and a single bias. These values are then passed on to the hidden layers where some of them are activated. In the backward pass, these values are modified once more by different weights and a single bias. Finally, when the values return to the input layer, the modified values are compared to the original input values. The goal of RBM is to repeat the training process until the original values and modified values are as close as possible. In this process a method called KL Divergence is used.

The key characteristic of RBM is that it can automatically find patterns in the data by reconstructing the input and decipher the interrelationship among input features. In addition, as RBM is a recreating process, the data does not have to be labelled. This is a big advantage as most datasets are not labelled.

H. Deep Belief Network

Deep Belief Networks (DBN) are formed by stacking RBMs. The structure of DBN is identical to the MLP, but the training process is totally different. DBN is a powerful tool as it overcomes the vanishing gradient problem.

This is how the training process works. DBN trains two layers at a time, and these two layers are treated just like an RBM. The first RBM is trained to reconstruct the input as accurately as possible. Then the hidden layer of the first RBM becomes the visible layer in the second RBM. The second RBM is trained by using the outputs of the first RBM. This process is repeated until every single layer in the DBN is trained.

A distinctive feature in DBN is that it works by fine-tuning the entire input in succession. This is how the vanishing gradient problem is solved. After the training is done, the model can now detect patterns in the data. However, labels are needed to interpret the meaning of the patterns. Fortunately, even a small set of labelled datasets are enough in improving the performance of the model.

III. Packages

There are a numerous number of packages that are released and used for building deep learning algorithms in R. A couple of frequently used packages and sample codes are introduced in the following.

A. R based

1. `deepnet`

Functionalities

`deepnet` is a package based on R only and was published on March 2014. Using the `deepnet` package, it is possible to implement various deep learning architectures and neural network algorithms, including BP (Back Propagation), RBM (Restricted Boltzmann Machine), DBN (Deep Belief Network), Deep Autoencoder and so on.

Example

This is a simple example of how to implement the functions in the `deepnet` package. First, load both the required package, and simple `iris` dataset containing 150 observations with 4 variables and 1 classification class. The `iris` data carries 50 observations of each of the 3 classes, which are `setosa`, `versicolor`, and `virginica`.

```
> library(deepnet)
> library(keras)
> data(iris)
```

To fix the random generators use the function `set.seed`. The input variables are assigned to `x` and the output variables are assigned to `y`. In order to get a more accurate model, scale the input variables using the min-max scaling method and split the dataset into two parts; train data and test data.

```
> set.seed(111)
> x <- as.matrix(iris[,1:4])
> maxs <- apply(x, 2, max)
> mins <- apply(x, 2, min)
> scaled_x <- as.matrix(scale(x, center = mins, scale = maxs - mins))
> y <- iris[,5]
> onehot_y <- to_categorical(as.numeric(y)-1)
# split the data (train data, test data)
> ind <- sample(nrow(iris), as.integer(nrow(iris)*0.7), replace = FALSE)
> train_x <- scaled_x[ind,]
> train_y <- onehot_y[ind,]
> test_x <- scaled_x[-ind,]
> test_y <- iris[-ind,5]
```

Now, train the data using the `dbn.dnn.train` function, which trains a deep neural network with weights and biases initialized by DBN. The results of the trained model show a high accuracy of over 97 percent.

```
> set.seed(111)
> dbn_dnn <- dbn.dnn.train(x = train_x, y = train_y,
+                          hidden = c(5, 5),
+                          output = 'softmax',
+                          numepochs = 50,
+                          cd = 10)
# predict the probability of the test data
> prob <- nn.predict(dbn_dnn, test_x)
> pred <- as.factor(apply(prob, 1, which.max))

> levels(pred) <- c('setosa', 'versicolor', 'virginica')

> table(pred, test_y)

              test_y
pred           setosa versicolor virginica
  setosa           18          0         0
  versicolor        0          9         0
  virginica         0          1        17

> sum(diag(table(pred, test_y)))/sum(table(pred, test_y))

[1] 0.9777778
```

This time using the `sae.dnn.train` function, it is possible to train a deep neural network with weights initialized by a stacked autoencoder.

```
> set.seed(211)

> sae_dnn <- sae.dnn.train(x = train_x, y = train_y,

+                         hidden=c(5, 5),

+                         numepochs = 50,

+                         output = 'softmax'

+                         )

# predict the probability of the test data

> prob <- nn.predict(sae_dnn, test_x)

> pred <- as.factor(apply(prob, 1, which.max))

> levels(pred) <- c('setosa', 'versicolor', 'virginica')

> table(pred, test_y)

              test_y
pred           setosa versicolor virginica
  setosa           18          0         0
  versicolor        0          9         0
  virginica         0          1        17

> sum(diag(table(pred, test_y)))/sum(table(pred, test_y))

[1] 0.9777778
```

2. NeuralNetTools

Functionalities

NeuralNetTools is a visualization and analysis tool for neural networks, that was published on July 2018. This package provides functions available for plotting, quantifying variable importance, obtaining a simple list of model weight and more.

Example

Using the same iris dataset as the example above, it is possible to visualize the trained neural network model. First, load the required visualization package NeuralNetTools, and neuralnet package which is used to train the dataset.

```
> library(NeuralNetTools)
> library(neuralnet)
> colnames(train_y) <- c('setosa', 'versicolor', 'virginica')
> binary_data <- data.frame(train_x, train_y)
```

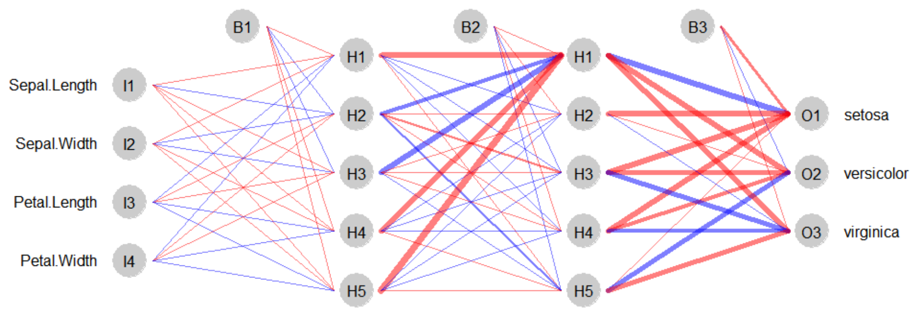
Before visualizing the neural net model, the model needs to be trained. The following code is the training process, which leads to a high test data accuracy of over 95 percent.

```
> set.seed(111)
> mod <- neuralnet(formula = setosa + versicolor + virginica ~
+                  Sepal.Length + Sepal.Width +
+                  Petal.Length + Petal.Width,
+                  data = binary_data,
+                  hidden = c(5, 5),
+                  rep = 10,
+                  err.fct = 'ce',
+                    linear.output = FALSE)
> prob <- predict(mod, train_x)
> pred <- as.factor(apply(p, 1, which.max))
> levels(pred) <- c('setosa', 'versicolor', 'virginica')
> sum(diag(table(pred, test_y)))/sum(table(pred, test_y))
[1] 0.9555556
```

Finally, using the function `plotnet`, the neural net model can be visualized.

```
> plotnet(mod_in = mod,
+          pos_col = 'blue',
+          neg_col = 'red',
+          circle_col = 'grey80',
+          bord_col = 'grey80',
+          alpha = 0.5)
```

Figure1. Visualization of neural net model



3. automl

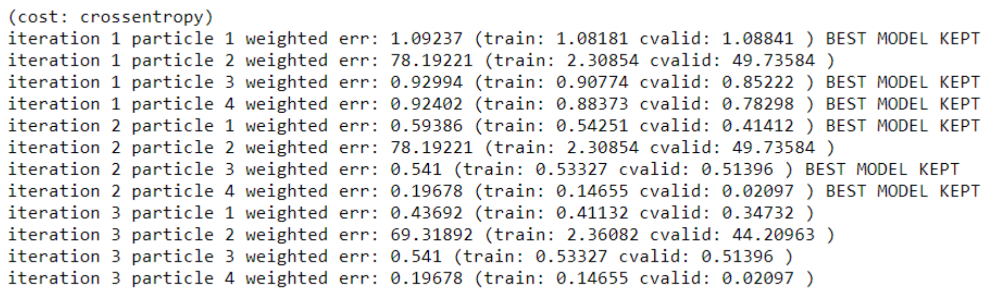
Functionalities

automl is a useful package that allows automatic machine learning of several deep neural networks with automatic hyperparameters tuning. The package tunes the model within the user-specified options and keeps updating the best model in the process. automl was published recently on March 2019.

Example

First, load the required automl package and train the automl model with a few user-specified options. The data used in this example is the same iris dataset as the previous examples.

```
> library(automl)
> amlmodel <- automl_train(Xref = train_x, Yref = train_y,
+                          autopar = list(numiterations = 3,
+                                         psopartpopsize = 4,
+                                         auto_modexec = TRUE,
+                                         seed = 118,
+                                         auto_minibatchsize_min = 6,
+                                         auto_layers_min = 1,
+                                         auto_layers_max = 3))
```



```
(cost: crossentropy)
iteration 1 particle 1 weighted err: 1.09237 (train: 1.08181 cvalid: 1.08841 ) BEST MODEL KEPT
iteration 1 particle 2 weighted err: 78.19221 (train: 2.30854 cvalid: 49.73584 )
iteration 1 particle 3 weighted err: 0.92994 (train: 0.90774 cvalid: 0.85222 ) BEST MODEL KEPT
iteration 1 particle 4 weighted err: 0.92402 (train: 0.88373 cvalid: 0.78298 ) BEST MODEL KEPT
iteration 2 particle 1 weighted err: 0.59386 (train: 0.54251 cvalid: 0.41412 ) BEST MODEL KEPT
iteration 2 particle 2 weighted err: 78.19221 (train: 2.30854 cvalid: 49.73584 )
iteration 2 particle 3 weighted err: 0.541 (train: 0.53327 cvalid: 0.51396 ) BEST MODEL KEPT
iteration 2 particle 4 weighted err: 0.19678 (train: 0.14655 cvalid: 0.02097 ) BEST MODEL KEPT
iteration 3 particle 1 weighted err: 0.43692 (train: 0.41132 cvalid: 0.34732 )
iteration 3 particle 2 weighted err: 69.31892 (train: 2.36082 cvalid: 44.20963 )
iteration 3 particle 3 weighted err: 0.541 (train: 0.53327 cvalid: 0.51396 )
iteration 3 particle 4 weighted err: 0.19678 (train: 0.14655 cvalid: 0.02097 )
```

Using the best model, it is possible to predict the class of the test data. The confusion matrix shows a high accuracy of over 95 percent.

```
> pred <- apply(automl_predict(model = amlmodel, X = test_x),
+               1, which.max)
> levels(pred) <- c('setosa', 'versicolor', 'virginica')
> sum(diag(table(pred, test_y)))/sum(table(pred, test_y))
[1] 0.9555556
```

4. autoencoder

Functionalities

The autoencoder package enables the implementation of the sparse autoencoder developed by Andrew Ng, and was published on June 2015. The features learned by the hidden layer of the autoencoder can be used in constructing deep belief neural networks. The visualization of the features learned by sparse autoencoder is also supported.

Example

Load the autoencoder package and set up the autoencoder architecture. Then train the sparse autoencoder with an unlabeled data using the function autoencoder. The mean error rate of the training data is 0.07.

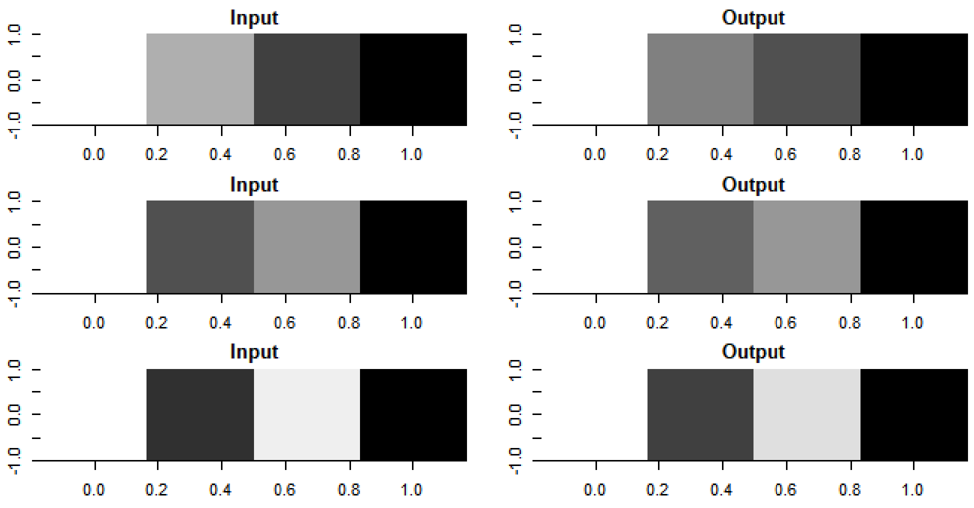
```
> library(autoencoder)
> nl = 3               ## number of layers
> unit.type = "tanh"  ## activation function ("logistic" or "tanh")
> Nx.patch = 1        ## width of training image patches, in pixels
> Ny.patch = 4        ## height of training image patches, in pixels
> N.hidden = 5        ## number of units in the hidden layer
> lambda = 0.0002      ## weight decay parameter
> beta = 6             ## weight of sparsity penalty term
> rho = 0.01           ## desired sparsity parameter
> epsilon <- 0.001     ## initialize weights
> max.iterations = 500 ## number of iterations in optimizer
> autoencoder.object <- autoencode(X.train = as.matrix(iris[,1:4]),
+                                  nl = nl, N.hidden = N.hidden,
+                                  unit.type = unit.type,
+                                  lambda = lambda, beta = beta,
+                                  rho = rho, epsilon = epsilon,
+                                  optim.method = "BFGS",
+                                  max.iterations = max.iterations,
+                                  rescale.flag = TRUE,
+                                  rescaling.offset = 0.001)
  autoencoding...
  Optimizer counts:
  function gradient
       172       86
  Optimizer: successful convergence.
  Optimizer: convergence = 0, message =
  J.init = 19.67715, J.final = 0.05203056, mean(rho.hat.final) = 0.01059616
> cat("autoencode(): mean squared error for training set: ",
+     round(autoencoder.object$mean.error.training.set,3),"\n")
autoencode(): mean squared error for training set:  0.07
```

Then using the trained model, predict the output and evaluate the performance of the model. Since the input and output image is analogous to one another, it can be said that the model is well trained.

```
> X.output <- predict(autoencoder.object,
+                     X.input = as.matrix(iris[,1:4]),
+                     hidden.output = FALSE)$X.output
```

```
## Compare outputs and inputs for 3 image patches
> op <- par(no.readonly = TRUE)
> par(mfrow=c(3,2), mar=c(2,2,2,2))
> for (n in c(1, 51, 101)){
+      ## input:
+      image(matrix(as.matrix(iris[,1:4])[n,],
+                   nrow = Ny.patch, ncol = Nx.patch),
+            axes = TRUE, main = "Input",
+            col = gray((0:32)/32))
+      ## output:
+      image(matrix(X.output[n,],
+                   nrow = Ny.patch, ncol = Nx.patch),
+            axes = TRUE, main = "Output",
+            col = gray((0:32)/32))
+ }
```

Figure2. Input image and Output image reconstructed by autoencoder



B. Python based

1. keras

Functionalities

keras is a high-level neural networks API developed with a focus on enabling fast experimentation. keras allows the code to be run on CPU or on GPU and is a user-friendly API which makes it easy to quickly prototype deep learning models. In addition, keras supports arbitrary network architectures and provides built-in support for convolution networks, recurrent networks, and any combination of both networks.

Example

First, load the keras package and set a random seed in order to ensure reproducible results using the function use_session_with_seed in the tensorflow package.

```
> library(keras)
> tensorflow::use_session_with_seed(1)
```

Then, initialize the model and build a deep neural network. In this example, the model has 2 hidden layers, each having 5 nodes, and a stochastic gradient decent with a learning rate of 0.1 is used as the optimizer. 20 percent of the train data is used as the validation data to find the best model. Then the model is trained with a batch size of 20 and 100 epochs.

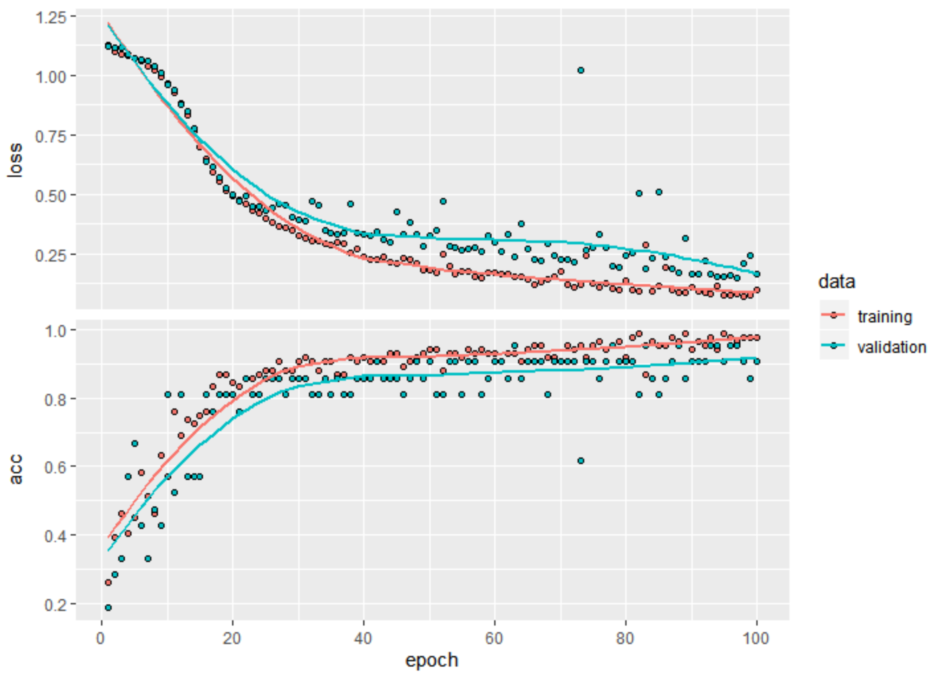
```
> model <- keras_model_sequential()
> model %>%
+       layer_dense(input_shape = ncol(train_x), units = 5) %>%
+       layer_dense(units = 5) %>%
+       layer_dense(units = ncol(train_y), activation = 'softmax’)
```

```
> summary(model)
```

```
________________________________________________________________________________
Layer (type)                        Output Shape                    Param #
================================================================================
dense (Dense)                       (None, 5)                       25
________________________________________________________________________________
dense_1 (Dense)                     (None, 5)                       30
________________________________________________________________________________
dense_2 (Dense)                     (None, 3)                       18
================================================================================
Total params: 73
Trainable params: 73
Non-trainable params: 0
________________________________________________________________________________
```

```
> sgd <- optimizer_sgd(lr = 0.1)

> model %>% compile(

+       loss = 'categorical_crossentropy',

+       optimizer = sgd,

+       metrics = 'accuracy')

> history <- model %>% fit(

+      x = train_x,

+      y = train_y,

+      epochs = 100,

+      batch_size = 20,

+      validation_split = 0.2,

+      verbose = 0)

> plot(history)
```

Figure3. Training process of dnn using keras



After training the model, test the performance of the model using the test data and the `predict_classes` function. The confusion matrix of the test data shows a high accuracy of 96 percent. The loss and accuracy can also be printed by the `evaluate` function.

```
> classes <- model %>% predict_classes(test_x)

> pred <- as.factor(classes)

> levels(pred) <- c('setosa', 'versicolor', 'virginica')

> table(pred, test_y)

               test_y
 pred           setosa versicolor virginica
   setosa           18          0         0
   versicolor        0         10         2
   virginica         0          0        15

> sum(diag(table(pred, test_y)))/sum(table(pred, test_y))

[1] 0.9555556
```

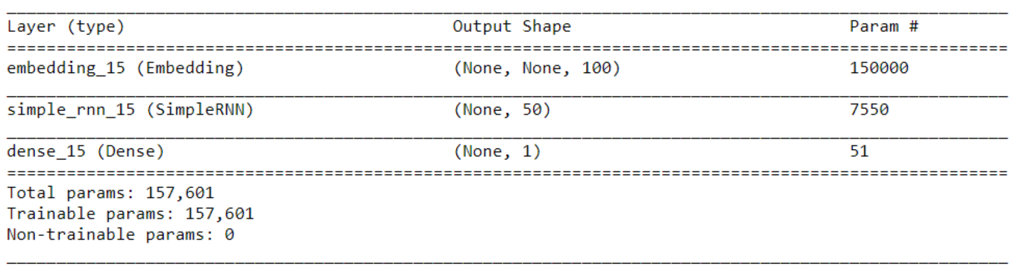
```
> model %>% evaluate(test_x, onehot_y[-ind,])
  $loss
  [1] 0.1020666

  $acc
  [1] 0.9555556
```

The `keras` package also has built-in RNN models that makes it easy to implement the RNN algorithm. This can be realized by using the `layer_simple_rnn` function when building the model. In the following example I will be using the IMDB movie review dataset, which is also included in the `keras` package. This dataset carries reviews from IMDB which have been preprocessed and encoded as a sequence of word indexes. The dataset consists a total of 25,000 reviews and each review is labeled by sentiment (positive/negative).

```
> max_unique_word <- 1500
> max_review_len <- 100
> my_imdb <- dataset_imdb(num_words = max_unique_word)
> x_train <- my_imdb$train$x
> y_train <- my_imdb$train$y
> x_test  <- my_imdb$test$x
> y_test  <- my_imdb$test$y
> x_train <- pad_sequences(x_train, maxlen = max_review_len)
> x_test <- pad_sequences(x_test, maxlen = max_review_len)
```

Now, build the RNN model using the `layer_simple_rnn` function. The RNN model has 50 nodes in the hidden layer and uses Adagrad as an optimizer. 80 percent of the train data will be trained with 15 epochs and then validated with 20 percent of the train data.

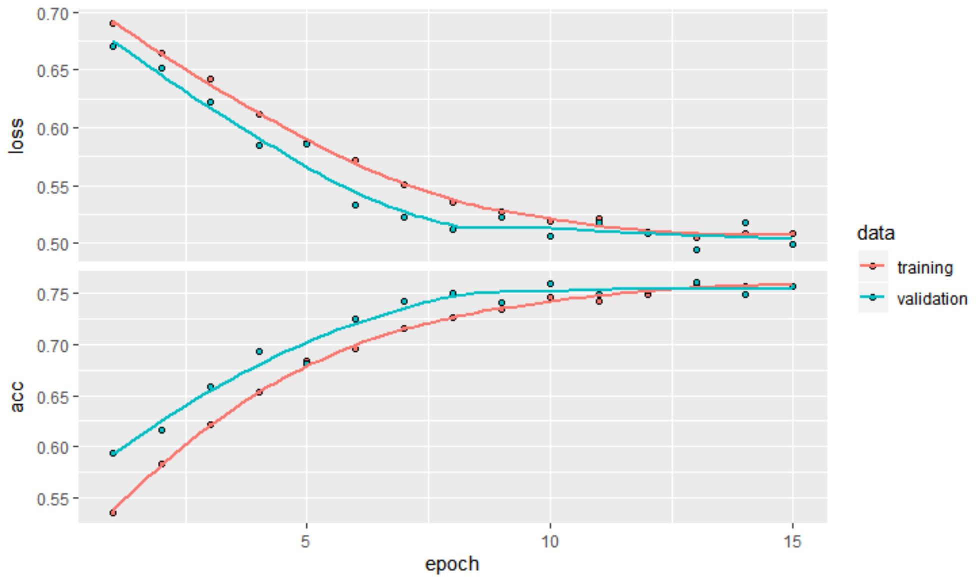
```
> rnn_model <- keras_model_sequential()
> rnn_model %>%
+     layer_embedding(input_dim = max_unique_word, output_dim = 100) %>%
+     layer_simple_rnn(units = 50, dropout = 0.2,
+                          recurrent_dropout = 0.2) %>%
+     layer_dense(units = 1, activation = 'sigmoid')
> summary(rnn_model)
```



```
________________________________________________________________________________________________________
Layer (type)                             Output Shape                        Param #
========================================================================================================
embedding_15 (Embedding)                 (None, None, 100)                   150000
________________________________________________________________________________________________________
simple_rnn_15 (SimpleRNN)                (None, 50)                          7550
________________________________________________________________________________________________________
dense_15 (Dense)                         (None, 1)                           51
========================================================================================================
Total params: 157,601
Trainable params: 157,601
Non-trainable params: 0
________________________________________________________________________________________________________
```

```
> rnn_model %>% compile(
+     loss = 'binary_crossentropy',
+     optimizer = 'adagrad',
+     metrics = c('accuracy'))
> rnn_history <- rnn_model %>% fit(
+     x_train, y_train,
+     batch_size = 100,
+     epochs = 15,
+     validation_split = 0.2)
> plot(rnn_history)
```

Figure4. Training process of rnn using keras

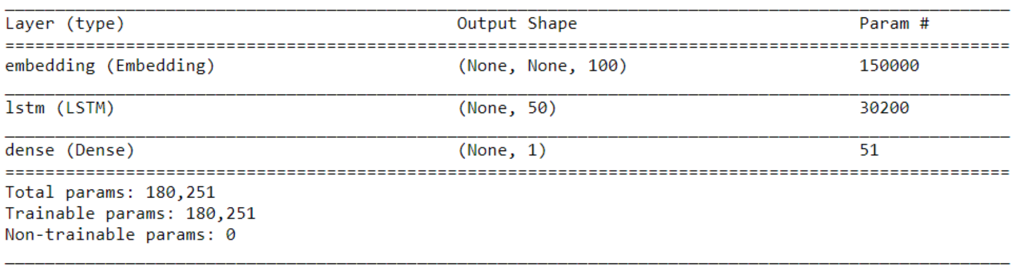


After training the model, evaluate the performance of the model with `predict_classes` and `evaluate` function. The test data shows an accuracy of over 76 percent. The performance is not very good due to the vanishing gradient problem in the training process.

```
> pred <- rnn_model %>% predict_classes(x_test)

> table(pred, y_test)

       y_test
pred     0    1
   0 9534 2946
   1 2966 9554

> sum(diag(table(pred, y_test)))/sum(table(pred, y_test))

[1] 0.76352

> rnn_model %>% evaluate(x_test, y_test)

25000/25000 [==============================] - 8s 322us/sample - loss: 0.4924 - acc: 0.7635
$loss
[1] 0.4924002

$acc
[1] 0.76352
```

In order to improve the performance of the model, this time train the model using LSTM. This can be done by simply switching the `layer_simple_rnn` function to `layer_lstm`. The accuracy has improved to over 83 percent, but at the same time the computation time has increased as well.

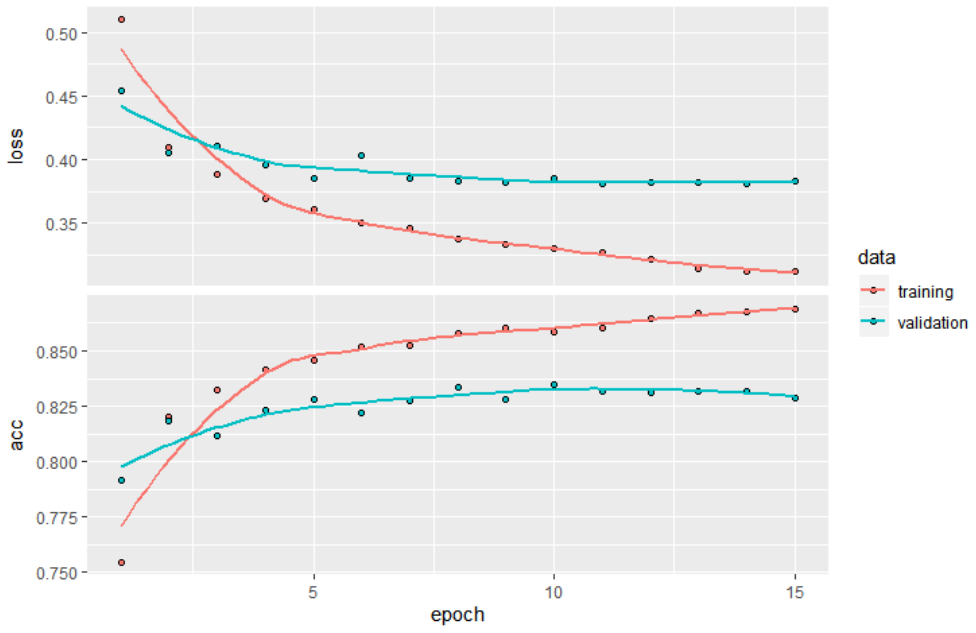
```
> lstm_model <- keras_model_sequential()
> lstm_model %>%
+       layer_embedding(input_dim = max_unique_word,
+                             output_dim = 100) %>%
+       layer_lstm(units = 50, dropout = 0.2,
+                       recurrent_dropout = 0.2) %>%
+       layer_dense(units = 1, activation = 'sigmoid')
> summary(lstm_model)
```



```
________________________________________________________________________________________
Layer (type)                              Output Shape                        Param #
========================================================================================
embedding (Embedding)                     (None, None, 100)                   150000
________________________________________________________________________________________
lstm (LSTM)                               (None, 50)                          30200
________________________________________________________________________________________
dense (Dense)                             (None, 1)                           51
========================================================================================
Total params: 180,251
Trainable params: 180,251
Non-trainable params: 0
________________________________________________________________________________________
```

```
> lstm_model %>% compile(
+      loss = 'binary_crossentropy',
+      optimizer = 'adagrad',
+      metrics = c('accuracy'))
> lstm_history <- lstm_model %>% fit(
+       x_train, y_train,
+       batch_size = 100,
+       epochs = 15,
+       validation_split = 0.2)
> plot(lstm_history)
```

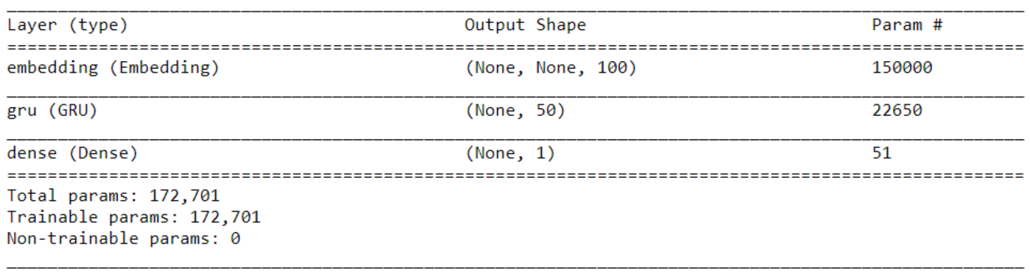
Figure5. Training process of lstm using keras



```
> pred <- lstm_history %>% predict_classes(x_test)

> table(pred, y_test)

      y_test
  pred     0     1
     0 10563  2235
     1  1937 10265

> sum(diag(table(pred, y_test)))/sum(table(pred, y_test))

[1] 0.83312

> lstm_history %>% evaluate(x_test, y_test)

25000/25000 [==============================] - 20s 810us/sample - loss: 0.3726 - acc: 0.8331
$loss
[1] 0.3725945

$acc
[1] 0.83312
```

In addition, training the model using GRU is also possible by simply switching the `layer_lstm` function to `layer_gru`. The accuracy of the model is 84 percent, slightly higher than LSTM.

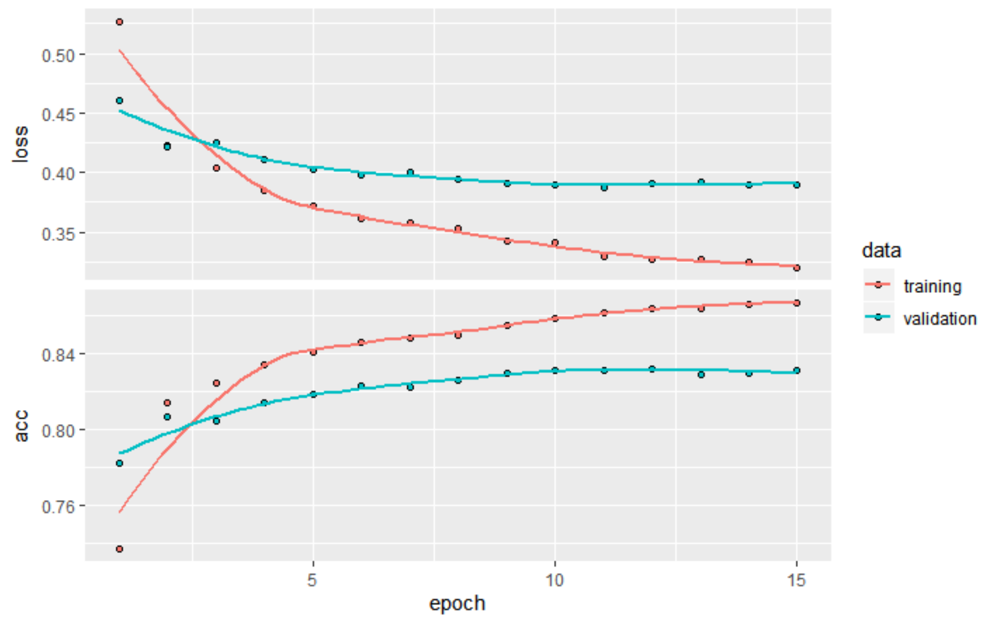
```
> gru_model <- keras_model_sequential()

> gru_model %>%

+     layer_embedding(input_dim = max_unique_word, output_dim = 100) %>%

+     layer_gru(units = 50, dropout = 0.2, recurrent_dropout = 0.2) %>%

+     layer_dense(units = 1, activation = 'sigmoid')

> summary(gru_model)
```



```
________________________________________________________________________________________
Layer (type)                          Output Shape                          Param #
========================================================================================
embedding (Embedding)                 (None, None, 100)                     150000
________________________________________________________________________________________
gru (GRU)                             (None, 50)                            22650
________________________________________________________________________________________
dense (Dense)                         (None, 1)                             51
========================================================================================
Total params: 172,701
Trainable params: 172,701
Non-trainable params: 0
________________________________________________________________________________________
```

```
> gru_model %>% compile(

+      loss = 'binary_crossentropy',

+      optimizer = 'adagrad',

+      metrics = c('accuracy'))

> gru_history <- gru_model %>% fit(

+      x_train, y_train,

+      batch_size = 100,

+      epochs = 15,

+      validation_split = 0.2)

> plot(gru_history)
```

Figure6. Training process of gru using keras



```
> pred <- gru_model %>% predict_classes(x_test)

> table(pred, y_test)

       y_test
   pred     0     1
      0 10526  2141
      1  1974 10359

> sum(diag(table(pred, y_test)))/sum(table(pred, y_test))

[1] 0.8354

> gru_model %>% evaluate(x_test, y_test)

25000/25000 [==============================] - 14s 559us/sample - loss: 0.3793 - acc: 0.8354
$loss
[1] 0.3792552

$acc
[1] 0.8354
```

C. Others

1. RcppDL

Functionalities

RcppDL is a package based on C++, published on December 2014. This package allows users to implement basic machine learning methods with many layers, such as DA (denoising Autoencoder), SDA (Stacked denoising Autoencoder), RBM (Restricted Boltzmann Machine) and DBN (Deep Belief Network).

Example

First, load the package RcppDL. As RBM is a method for reconstructing the given dataset, the whole iris data will be used to fit the model instead of just the train data. The function Rrbm returns an object for RBM, and the function reconstruct reconstructs the dataset. The results show a MSE value of 0.512

```
> library(RcppDL)
# train restricted boltzmann machine
> set.seed(1234)
> rbm_train <- Rrbm(x = scaled_x)
> train(rbm_train)
# reconstruction rate
> rbm_recon <- reconstruct(rbm_train, test = scaled_x)
> sqrt(mean((rbm_recon - scaled_x)^2))
[1] 0.5119204
```

This time using the Rda function in the RcppDL package, it is possible to reconstruct the iris dataset with a denoising autoencoder model. The results show a MSE value of 0.482.

```
> set.seed(123)
> ae_train <- Rda(x = scaled_x)
> train(ae_train)
# reconstruction rate
> ae_recon <- reconstruct(ae_train, test = scaled_x)
> sqrt(mean((ae_recon-scaled_x)^2))
[1] 0.4823496
```

2. `mxnet`

Functionalities

`mxnet` is a flexible and efficient tool for computing deep learning which enables users to construct and customize the state-of-art deep learning models in R. This package is based on C++ and was first published on June 2017.

Example

Load the `mxnet` package and fix the random number using the function `mx.set.seed`. If installing the `mxnet` package does not work, try the following code.

```
> install.packages("https://s3.ca-central-1.amazonaws.com/jeremiedb/share/mxnet/CPU/3.6/mxnet.zip", repos = NULL)
> install.packages("DiagrammeR")
```

In this example, we will be using the original data for training the model as the original data fits better than the scaled data. Keep in mind that the input data needs to be in the form of a matrix and only an array is supported for the training label. Then, customize your model using various options supported in the `mx.mlp` function.

```
> library(mxnet)
> mx.set.seed(0)
> mx_train_x <- data.matrix(iris[ind,1:4])
> mx_train_y <- as.numeric(y[ind]) – 1
> model <- mx.mlp(data = mx_train_x,
                      label = mx_train_y,
                      hidden_node = c(10, 10),
                      out_node = 3,
                      out_activation = "softmax",
                      num.round = 100,
                      array.batch.size = 10,
                      learning.rate = 0.05,
                      momentum = 0.8,
                      eval.metric = mx.metric.accuracy,
                      array.layout = "rowmajor")
```

After training the model, test the performance of the model using the test data and the `predict` function. The confusion matrix of the test data shows a high accuracy of 84 percent.

```
> mx_test_x <- data.matrix(iris[-ind,1:4]) - 1
> mx_test_y <- y[-ind]
> pred <- predict(model, mx_test_x, array.layout = "rowmajor")
> pred.label <- as.factor(max.col(t(pred)))
> levels(pred.label) <- c('setosa', 'versicolor', 'virginica')
> table(pred.label, mx_test_y)
```

```
              mx_test_y
pred.label   setosa versicolor virginica
  setosa         18          0         0
  versicolor      0         10         7
  virginica       0          0        10
```

```
> sum(diag(table(pred.label, mx_test_y)))/sum(table(pred.label, mx_test_y))
[1] 0.8444444
```

3. h2o

Functionalities

h2o is a R interface for 'H2O', a scalable open source machine learning platform that offers both supervised and unsupervised machine learning algorithms. Deep neural networks are also supported, and this package was published on August 2019.

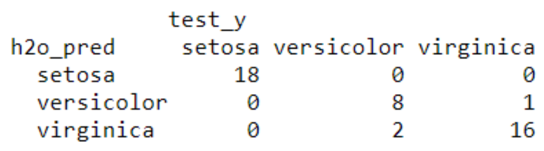
Example

First load the h2o package and set an active connection by using the h2o.init function. It is possible to build a feed-forward multilayer neural network using the h2o.deeplearning function. In the example code, I will train the model consisting 2 hidden layers with the original iris dataset.

```
> library(h2o)
> h2o.init()
> iris_frame <- as.h2o(iris[ind,])
> iris_test <- as.h2o(iris[-ind,])
> model <- h2o.deeplearning(x = 1:4,
+                           y = 5,
+                           training_frame = iris_frame,
+                           activation = "RectifierWithDropout",
+                              seed = 123,
+                              hidden = c(5,5),
+                              epochs=100)
```

After training the model, test the performance of the model using the test data and the `h2o.predict` function. The confusion matrix of the test data shows a high accuracy of 93 percent.

```
> h2o_pred <- h2o.predict(model, iris_test)
> h2o_pred <- as.vector(h2o_pred$predict)
> table(h2o_pred, test_y)
```



```
            test_y
h2o_pred      setosa versicolor virginica
  setosa         18          0         0
  versicolor      0          8         1
  virginica       0          2        16
```

```
> sum(diag(table(h2o_pred, test_y)))/sum(table(h2o_pred, test_y))
[1] 0.9333333
```

D. Summary

Table1. Summary of R packages

	`deepnet`	**`NeuralNetTools`**	**`automl`**	**`autoencoder`**	**`keras`**	**`RcppDL`**	**`mxnet`**	**`h2o`**
based	R	R	R	R	Python	Rcpp	Rcpp	Rcpp
DNN	O		O	O	O		O	O
CNN					O		O	
RNN					O		O	
LSTM					O			
GRU					O			
AE	O			O		O		
RBM	O					O		
DBN	O					O		
viz.		O		O			O	

The table above is the summary of the R packages handled in this paper. It shows what kind of structure each R package can build up. We can see that almost all deep learning packages support the basic DNN model. The `keras` package provides the most diverse types of deep learning structures and the algorithms are very easy to implement. The reconstruction of the input data can be done by packages `deepnet`, `autoencoder`, and `RcppDL`. Packages such as `NeuralNetTools`, `autoencoder`, `mxnet` specialize in visualizing the structure of the model or the trained filters.

IV. Application

A. Data Description

The data used in the paper is a bank marketing data related with direct marketing campaigns of Portuguese banking institution. The collected data is from May 2008 to November 2010 and is consisted of 45,211 instances and 17 variables. The variables are made up of the followings; age, job, marital status, education status, default, average balance, housing loan status, personal loan status, contact type, last contact day, last contact month, contact duration, number of contacts during campaign, number of days passed after the last contact, number of contacts before campaign, outcome of previous marketing campaign. The target variable is a binary variable indicating whether the client has subscribed a term deposit or not.

Table2. Description of variables

Variables	**Description**
age	integer between 18 and 95
job	type of job; admin, management, housemaid, entrepreneur, student, blue-collar, self-employed, retired, technician, services, unemployed, unknown
marital	marital status; married, divorced (or widowed), single
education	primary, secondary, tertiary, unknown
default	credit default status; no, yes
balance	average yearly balance, in euros; integer between -8,018 and 102,127
housing	housing loan status; no, yes
loan	personal loan status; no, yes
contact	contact communication type; cellular, telephone, unknown
day	last contact day of the month; integer between 1 and 31
month	last contact month of year; jan, feb, mar, apr, may, jun, jul, aug, sep,

	oct, nov, dec
duration	last contact duration, in seconds; integer between 0 and 4,918
campaign	number of contacts performed during this campaign and for this client; integer between 1 and 63
pdays	number of days that passed by after the client was last contacted from the previous campaign; integer between -1 (client was not previously contacted) and 871
previous	number of contacts performed before this campaign and for this client; integer between 0 and 275
poutcome	outcome of the previous marketing campaign; failure, success, other, unknown
y	whether or not the client has subscribed a term deposit; no, yes

B. Data Analysis

Using the packages mentioned above, we will build various types of deep neural networks and compare the performance of each model.

1. Data Preprocessing

The bank data is consisted of 45,211 observations but the degree of imbalance is severe; ‘yes’ takes up approximately 12% whereas ‘no’ takes up approximately 88%. In order to improve the performance of the model, we will be using the under-sampling method. This forms a smaller dataset with a total of 10,593 observations composed of almost the same proportion of the y variable.

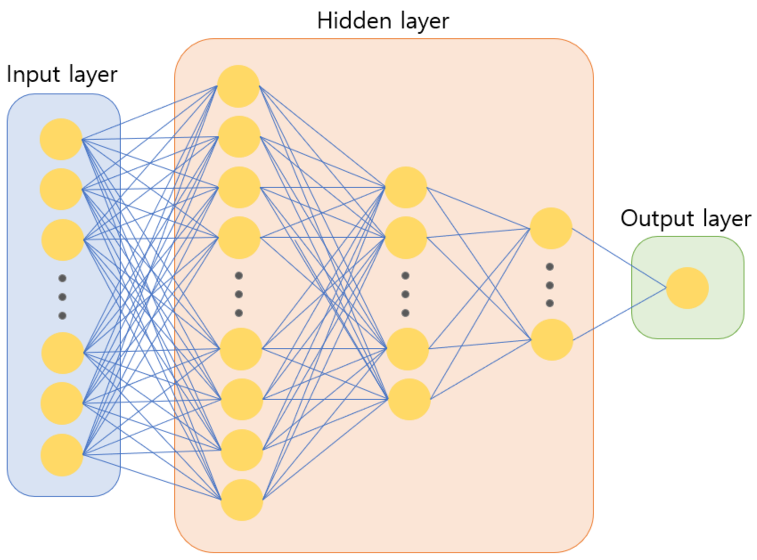
When training a neural network model, it is important to remember that categorical variables need to be in the form of one-hot encoding. The bank data has 7 numerical variables and 9 categorical variables. The target variable y is also a binary categorical variable. One-hot encoding can be easily implemented by using the `to_categorical` function provided by the `keras` package. In addition, scaling the numerical variables might be of help in dramatically increasing the performance of the model depending on the dataset. When training the bank data, we will scale the numerical variables using the min-max scaling method.

Once the one-hot encoding of categorical variables and scaling of numerical variables are done, we are ready to split the dataset into two parts; the train data and test data. The transformed bank dataset is now consisted of 10,593 observations and 44 variables. Among the total dataset, we will use 70 percent of the data as the train data and the remaining 30 percent as the test data.

2. Modeling

We will be using a DNN model to compare the performance of various R packages. The DNN model has 3 hidden layers with 20, 40, 10 nodes in each layer. In the following we will train the same model with various R packages and compare the accuracy and computation time of each method.

Figure7. DNN model



3. Results

First, we trained the data with an identical DNN model. Using the `deepnet` library we will train the DNN model with functions `dbn.dnn.train` and `sae.dnn.train`. These models are pretrained each with an DBN algorithm and a SAE algorithm and then trained by a DNN model. The `automl`, `keras`, `mxnet`, and `h2o` packages all train the data using the DNN model without a pretraining process. The results of the accuracy and computation time of each package is organized in the following table.

Table3. Results of Portuguese bank data with DNN model

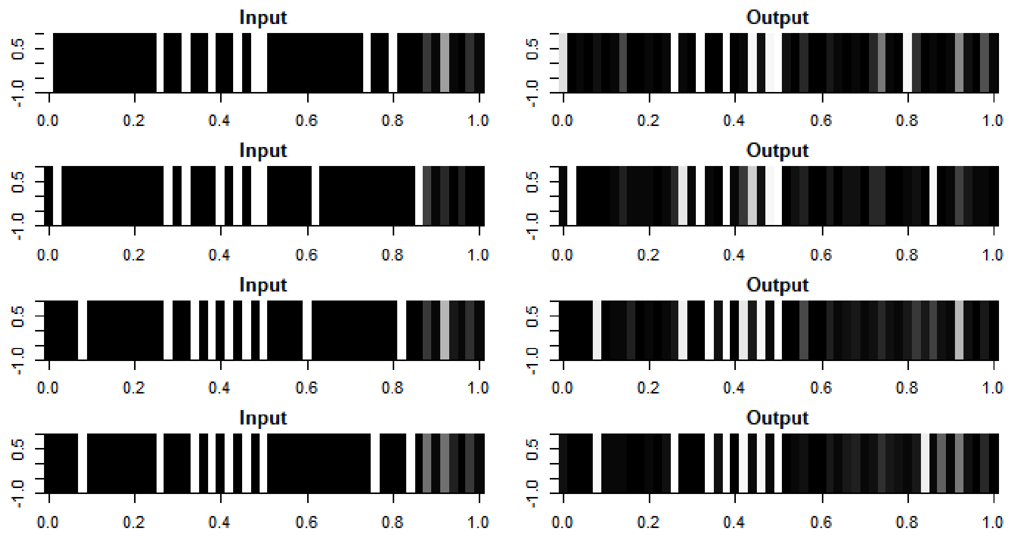
package	**`deepnet`**		**`automl`**	**`keras`**	**`mxnet`**	**`h2o`**
model	DBN, DNN	SAE, DNN	DNN	DNN	DNN	DNN
accuracy (%)	72.5	85.2	80.3	81.2	59.8	88.6
computation time (s)	46.35	8	433.8	8.25	22.53	1.16

According to the results of DNN, the model trained using the `h2o` package has the highest accuracy and shortest computation time. On the contrary, `automl` has the longest computation time leading up to over 7 minutes. This is due to the heavy calculation in the process of automatically finding the best model within the given restrictions. Except for the results of `mxnet`, the overall accuracy of the DNN model is about 80 percent.

Next, using the `RcppDL` package we will reenact the input data. The `RcppDL` package focuses on reconstructing the input data, so the results of this package returns the reproduced value of the original data. In order to check the performance of the pretraining process, we calculated the RMSE value of the input data and regenerated data. The RMSE of the data is 0.3113, which means that the original data is reproduced well.

Finally, the `autoencoder` library is used to compare how well the provided function can reconstruct the input data, and the results are provided visually. The first two images are input values of clients who do not subscribe a term deposit, and the last two images are input values of clients who subscribe a term deposit. The input and output images are analogous, which means that the autoencoder is well trained.

Figure8. Visualization of autoencoder



V. Conclusion

Using the Portuguese marketing bank data, we compared the accuracy and computation time between various R packages. Looking into the result, we can see that the `h2o` package shows the best performance; the accuracy is the highest and running speed is the fastest. This is because the `h2o` package is based on JAVA and the entire process is handled in the h2o cluster environment. The rest of the packages all show a similar level of performance except for the `mxnet` package. The reason `mxnet` shows a poor performance is thought to be because it is based on Rcpp. To sum up, in the case of modeling the bank data using a DNN model, it can be said that the `h2o` package is the best package.

The best model and the optimal R package for learning the data depends on the type of data, so it is difficult to pinpoint one particular method. However, since the `keras` package supports the most diverse algorithms and the overall performance is good, we recommend using this package for basic analysis. The `automl` is convenient as it automatically finds the best model, but it has its drawback in that the computation time is long. Improving the performance of the model is also possible by pretraining the model with the reconstruction process such as AE, SAE, RBM, DBN, and this can be done with the `deepnet` and `RcppDL` package. In addition, visualizing tools are provided in `NeuralNetTools`, `autoencoder`, and `mxnet` packages.

As deep learning is getting more popular and commercialized in a lot of fields, the demand for information in how to implement deep learning is growing. Recently, various methods using R have appeared and this paper introduced some of most frequently used R packages. In the future, we expect more advanced and sophisticated deep learning R packages to appear, and that more people will use R when analyzing their data.

Bibliography

[1] Arnold, T. B. (2017). kerasR: R Interface to the Keras Deep Learning Library. *J. Open Source Software*, *2*(14), 296.

[2] Beck, M. W. (2018). NeuralNetTools: Visualization and analysis tools for neural networks. *Journal of statistical software*, *85*(11), 1.

[3] Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, *5*(2), 157-166.

[4] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2015, June). Gated feedback recurrent neural networks. In *International Conference on Machine Learning* (pp. 2067-2075).

[5] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

[6] Dubossarsky, E., & Tyshetskiy, Y. autoencoder: Sparse Autoencoder for Automatic Learning of Representative Features from Unlabeled Data, r package version 1.1 (2015). *URL https://CRAN. R-project. org/package= autoencoder*.

[7] Fischer, A., & Igel, C. (2012, September). An introduction to restricted Boltzmann machines. In *Iberoamerican Congress on Pattern Recognition* (pp. 14-36). Springer, Berlin, Heidelberg.

[8] Fu, A., Aiello, S., Rao, A., Kraljevic, T., Maj, P., Kraljevic, M. T., & Java, S. (2014). Package ‘h2o’.

[9] Gijsbers, P., LeDell, E., Thomas, J., Poirier, S., Bischl, B., & Vanschoren, J. (2019). An open source AutoML benchmark. *arXiv preprint arXiv:1907.00909*.

[10] Hodnett, M., & Wiley, J. F. (2018). *R Deep Learning Essentials: A step-by-step guide to building deep learning models using TensorFlow, Keras, and MXNet*. Packt Publishing Ltd.

[11] Hothorn, T. (2019). CRAN task view: Machine learning & statistical learning.

[12] Hua, Y., Guo, J., & Zhao, H. (2015, January). Deep belief networks and deep learning. In *Proceedings of 2015 International Conference on Intelligent Computing and Internet of Things* (pp. 1-4). IEEE.

[13] LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *nature*, *521*(7553), 436.

[14] Montavon, G., Samek, W., & Müller, K. R. (2018). Methods for interpreting and understanding deep neural networks. *Digital Signal Processing*, *73*, 1-15.

[15] [Moro et al., 2014] S. Moro, P. Cortez and P. Rita. A Data-Driven Approach to Predict the Success of Bank Telemarketing. Decision Support Systems, Elsevier, 62:22-31, June 2014

[16] Ripley, B. (2013). Feed-Forward Neural Networks and Multinomial Log-Linear Models [R package nnet version 7.3-12].

[17] Rong, X. (2014). deepnet: deep learning toolkit in R. *R package version 0.2. URL: http://CRAN. R-project. org/package= deepnet*.

국문초록

박정민

통계학과

이화여자대학교 대학원

오늘날 딥러닝은 생물정보학, 의료영상학, 금융공학, 컴퓨터 과학 등 거의 모든 분야에서 각광받고 있다. 대부분의 딥러닝 알고리즘은 Python으로 짜여져 있는데 이는 백엔드 텐서 계산방식이 Tensorflow를 기반으로 하기 때문이다. 그렇지만 최근 들어서는 통계 분야에서 더 활발하게 사용되는 R에서도 딥러닝의 구현이 가능하게 되었다. 이 논문에서는 R에서 사용할 수 있는 다양한 딥러닝 패키지를 소개하고 각 패키지의 사용법에 대한 가이드라인을 제공한다.

먼저, 딥러닝의 다양한 알고리즘을 간단히 소개하고 딥러닝 학습을 위한 최첨단 소프트웨어를 검토한다. 본 논문에서 소개할 R 패키지는 다음과 같다 – **deepnet, NeuralNetTools, automl, autoencoder, keras, RcppDL, mxnet, h2o**. 또한, 사용자들이 각 패키지의 기능을 쉽게 이해할 수 있도록 샘플코드를 제공하고 포르투갈의 은행 데이터를 활용하여 다양한 R 함수의 성능을 각각 비교하는 것으로 논문을 마무리한다.