# Ruta: implementations of neural autoencoders in R

David Charte[a], Francisco Herrera[a], Francisco Charte[b]

[a] *Andalusian Research Institute on Data Science and Computational Intelligence (DaSCI), University of Granada, 18071 - Granada, Spain*
[b] *Andalusian Research Institute on Data Science and Computational Intelligence (DaSCI), University of Jaén, 23071 - Jaén, Spain*

**Abstract**

Autoencoders are neural networks which perform feature learning on data. Many variants can be found in the literature, but their implementations are scarce, in separate software pieces and utilizing different languages and frameworks. The `ruta` package implements a unified foundation for the construction and training of autoencoders on top of Keras and Tensorflow, and allows for easy access to the main functionalities as well as full customization of their diverse aspects.

*Keywords:* unsupervised learning, neural networks, autoencoders
*2010 MSC:* 62M45, 68T01

## 1. Introduction

The problem of feature extraction consists in finding a transformation of the feature space of some data set which is more adequate than the original one in relation to another task, such as classification or visualization. A particular case of this problem is dimensionality reduction, where the objective is to build a more compact representation for the data while retaining most of their information.

Some traditional techniques for feature extraction are principal components analysis (PCA) [1], multidimensional scaling [2], Isomap [3] and locally linear embedding [4]. Other more modern methods include t-distributed stochastic neighbor embedding (t-SNE) [5], which is designed to visualize

*Email addresses:* `fdavidcl@ugr.es` (David Charte), `herrera@decsai.ugr.es` (Francisco Herrera), `fcharte@ujaen.es` (Francisco Charte)

high-dimensional datasets, restricted Boltzmann machines (RBMs) [6] and autoencoders (AEs) [7], both based on neural networks.

AEs are a tool for feature extraction in increasing development. Making use of them, however, is not straightforward. Software pieces which implement them are uncommon and are either very basic versions or adapted to specific databases. Basic AE models are relatively easy to implement in well-known deep learning frameworks, such as Keras [8] or Tensorflow [9], but this requires some knowledge about their structure and training procedures. In addition to this, some useful regularizations and alterations in the objective functions can present challenges while coding. Since most neural AEs share a common basis, it is desirable to have an implementation which abstracts its components and gives the customization possibilities to build different kinds of AEs without reimplementing them. This would allow users to leverage the possibilities of AEs as feature learning techniques without the need to study their architecture in advance.

The `ruta` package for the R language includes all the necessary foundations to build AEs for all kinds of experimentations. It is based on frameworks Keras and Tensorflow to ensure efficiency and cross-platform compatibility. Its interface allows any R user to easily define different models, train them and perform additional tasks with little to no previous knowledge required.

## 2. Problems and Background

As previously stated, the main objective of an AE is to find a good transformation of the features according to one or more criteria. When an instance is mapped to the new feature space, it is seen as an *encoding* of the original. This encoding must allow the AE to reconstruct the instance from the original feature space by means of a decodification process. Intuitively, this reconstruction can only be achieved if sufficient information about each instance is retained within the encoding.

### *2.1. Autoencoder framework*

An AE [10] is an artificial neural network (ANN) composed of an encoder and a decoder. Analytically, it can be seen as a composition of maps $f$ and $g$ which results in a tensor of the same shape as the input. As an ANN, it takes a form analogue to that on Fig. 1. AEs were originally used to perform a preliminary weight training on other ANNs, but on their own they can also learn alternative representations for input data.

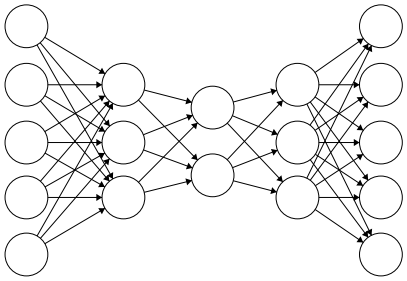

Figure 1: A possible neural architecture for an AE with a 2-variable encoding layer.

The different aspects that lead an AE to a specific transformation are its neural architecture, which determines the type of input and the size of the encoding; the cost and activation functions, which can be defined and regularized in order to induce some desired properties, and parameters of the training process, such as the optimization algorithm or the number of times the data is feeded to the network.

### *2.2. Variants*

An interesting advantage of AEs is their versatility: one can obtain encodings with certain properties if the adequate regularizations are chosen. There exist many AE variants in the literature [10], the most common ones centered in how to control the behavior of the transformation while allowing for faithful reconstructions. The following are the most relevant ones:

- Sparse: induces a low number of activations in average in the encoding layer.
- Contractive: attempts to preserve the local structure of the original space, thus searching for coordinates in a lower-dimensional manifold.
- Denoising: is able to remove noise introduced in input examples.
- Robust: is less sensitive to noise in instances due to a different loss function.
- Variational: extracts a generative model from the data and is able to produce new, unseen instances.
- Adversarial: trains in an adversarial manner with the aim of forcing the encoding to follow a given distribution.

- Convolutional and LSTM-based: are composed of other types of units and layers in order to accomodate bidimensional and sequential data, respectively.

## 3. Software Framework

In this section we elaborate on the internal structure of the developed software and its functionality.

### *3.1. Software Architecture*

The object system utilized in `ruta` is S3, a minimal object orientation from the R language based on generic functions. The software is developed around several classes which have certain applicable methods:

- `ruta_autoencoder`: represents a parametrized AE learner. It can be trained and can perform several post-train tasks, such as data encoding and reconstruction.
- `ruta_network`: defines neural network structures by layers. Networks can be concatenated to produce a longer one.
- `ruta_loss`: represents the loss function to be optimized by the learner. It is either a wrapper over a loss function from Keras, or a built-in loss function such as correntropy.
- `ruta_noise`: represents a type of noise which can be applied to input data. Several of these are provided within the package for convenience.

### *3.2. Software Functionalities*

The main functionalities of package `ruta` are as follows:

- Define and customize diverse aspects of an AE model.
- Train AE variants according to the desired objective function.
- Encode and reconstruct input data with a trained model.
- Evaluate a trained model according to several metrics which account for quality of reconstruction.
- Sample generative models created by variational AEs.

- Generate corrupted data with different types of noise.

The programming interface provided by the package gives several ways to access this set of functionalities, according to the desired level of customization and difficulty:

- Directly train an AE and compress a database via function `autoencode`.
- Define a basic AE simply by enumerating the dimensions of its layers in a vector, e.g. `autoencoder(c(32, 6))`.
- Define each layer composing the neural architecture by means of functions `input`, `dense`, `conv`, `output`, etc., then construct an AE with possibly one or more variant properties.

The following AE types can be used: basic, sparse, contractive, denoising, robust, variational and convolutional (via the included `conv` layers). Some of them may be combined by means of the `make` family of functions, e.g. `make_sparse`. They are extensively documented within the package and in the online documentation[1].

### *3.3. Implementation Details*

Since `ruta` is implemented on top of Tensorflow and Keras, it can run on computing devices such as GPUs. In order for them to be used, the correct Tensorflow version with CUDA support will need to be installed. Several issues can arise during the installation and first use, which have been documented in the troubleshooting section of the online documentation.

Few other software pieces provide the necessary functionality to build custom AEs. Among them we can find H2O [11], with its `h2o.deeplearning` function which includes an `autoencoder` option; package `autoencoder` for R [12], and library `yadlt` for Python. These focus on just one or two AE variants and provide less customizability than AEs defined in `ruta`. For further options one needs to resort to Deep Learning frameworks, which require a much higher programming effort in order to define AE models.

[1] `https://ruta.software`

## 4. Illustrative Examples

An easy way to start using `ruta` is by means of the `autoencode` function. This will take a dataset and automatically train a simple AE and produce a codification for it. The function accepts several parameters, from which only the desired dimension is mandatory. Other optional parameters are the type of AE, the activation function in the middle layer and the number of epochs for the training process. The following example uses this function to extract 2 features from the well-known toy dataset Iris:

```
library(ruta)
library(purrr)

encoded <- iris[, 1:4] %>% as.matrix() %>% autoencode(2, "robust")
```

These 2 features can be visualized like in Fig. 2 in order to represent the model learned by the AE.

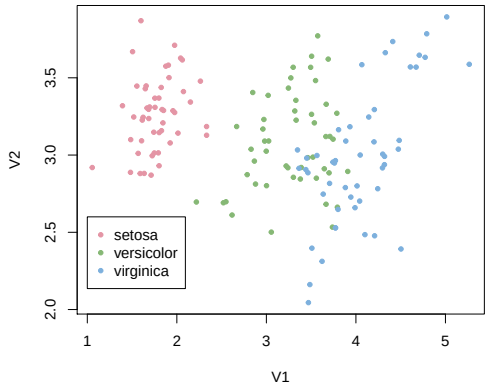


Figure 2: Features learned by a basic AE with Iris data.

The next step in difficulty involves defining a deep autoencoder. To help beginners describe its architecture, `ruta` provides a conversion from integer vector to neural network architecture in the following manner: `c(64, 16)` would become a network with an input layer the size of the inputs, a hidden layer with 64 variables, another hidden layer with 16 units for the encoding, the last hidden layer with 64 variables and an output layer the same size of the input one. Thus, the interface allows for simpler code, which can be

observed in the following comparison between the code needed to define the same model in `ruta` and Keras:

```
xtrain <- quakes[1:750,] %>% as.matrix()
xtest <- quakes[751:1000,] %>% as.matrix()
code_dim <- 2
hidden_dim <- 6

# ============== Ruta  ==============
features <- autoencoder(c(hidden_dim, code_dim), "sigmoid") %>%
  train(xtrain) %>%
  encode(xtest)

# ============== Keras ==============
input_l <- layer_input(shape = 5)
encoded <- layer_dense(input_l, units = hidden_dim)
encoded <- layer_dense(encoded, units = code_dim, activation = "sigmoid")
decoded <- layer_dense(encoded, units = hidden_dim)
decoded <- layer_dense(decoded, units = 5)

autoe <- keras_model(input_l, decoded)
encoder <- keras_model(input_l, encoded)
compile(autoe, loss = "mean_squared_error", optimizer = "rmsprop")
fit(autoe, xtrain, xtrain)
features <- predict(encoder, xtest)
```

The following example loads a dataset from Keras and normalizes its variables. Afterwards it defines a sparse AE by means of the `autoencoder_sparse` function with a 3-variable encoding, trains it and uses it to reconstruct test data. An evaluation is performed according to the mean squared error metric for the same test data.

```
boston <- keras::dataset_boston_housing()

train_x <- scale(boston$train$x)
test_x <- scale(
  boston$test$x,
  center = train_x %@% "scaled:center",
  scale = train_x %@% "scaled:scale"
)

learner <- autoencoder_sparse(
  input() + dense(3, "tanh") + output(),
  "mean_squared_error"
)
model <- train(learner, train_x, epochs = 200)

reconstructions <- reconstruct(model, test_x)
evaluate_mean_squared_error(model, test_x)
```

Another task that can be performed by a trained variational AE is gen-

eration of new instances. In this case, we load the MNIST dataset of handwritten digits and learn 10 features which can be sampled via the `generate` function. Instances can also be generated by interpolating encodings from existing instances and decoding those interpolations, as Fig. 3 shows.

```
mnist = keras::dataset_mnist()

x_train <- keras::array_reshape(
  mnist$train$x, c(dim(mnist$train$x)[1], 784)
) / 255.0
x_test <- keras::array_reshape(
  mnist$test$x, c(dim(mnist$test$x)[1], 784)
) / 255.0

network <-
  input() +
  dense(256, "elu") +
  variational_block(10, seed = 42) +
  dense(256, "elu") +
  output("sigmoid")
learner <- autoencoder_variational(network, loss = "binary_crossentropy")
model <- train(learner, x_train, epochs = 10)

samples <- model %>% generate(dimensions = c(8, 5), side = 6, fixed_values =
    0.99)
```

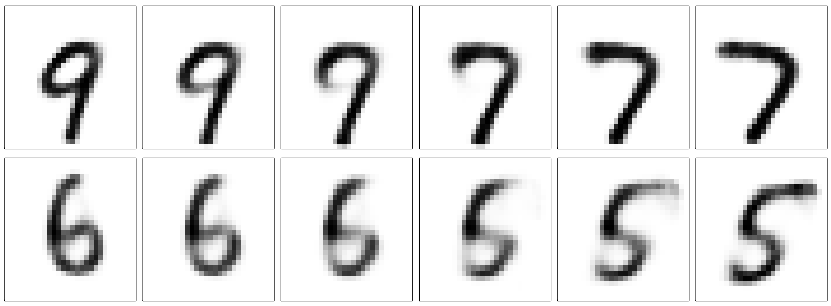

Figure 3: Instances generated when interpolating between test samples in a variational AE trained with MNIST data.

The generic AE templates provided within the package may not always be adaptable enough for some problems. Thus, in order to provide detailed control over the model for more advanced users with some knowledge of Keras, `ruta` can convert its AE objects into a list of Keras models. This list contains three models: one for the encoder, another one for the decoder and one for the full AE. It can be accessed by setting the input shape in the Ruta object and calling the `to_keras` method:

```
obj <- autoencoder_contractive(c(128, 16))
obj$input_shape <- 1000
models <- to_keras(obj)
print(models$autoencoder)
```

Individual examples for each AE type are provided in the online documentation, as well as detailed instructions on how to build more customized neural architectures.

## 5. Conclusions

In this paper, we have presented a novel software piece focused in the construction of AEs, the `ruta` package for R. As opposed to most software developed on this topic, `ruta` implements several well-known AE variants and can handle different datasets. The software is implemented on top of Tensorflow and Keras in order to provide good performance, but abstracts many common aspects of AEs in order to provide an easy-to-use interface, accessible to R users with or without a programming background.

We have provided examples on how trained AEs can perform several tasks such as encoding and reconstruction of new data, as well as evaluation and even instance generation. When users need more control over the automatic generation of AE architectures, the package allows to extract the associated Keras models so as not to hinder their customization.

Since its publication on CRAN in May 2018 to the end of the year, `ruta` has received more than a thousand downloads from the RStudio CRAN mirror. Fig. 4 shows the amount of downloads since the day of publication.

Some supplementary software packages have already been planned. These include a package dedicated to visualizing the behavior of AEs, from their training process to the learned model, and a web-based user interface with the aim of providing easier access to these neural architectures.

## Acknowledgements

This work was partially funded by the grant Iniciación a la Investigación para Alumnos de Máster of the University of Granada, project TIN2015-68854-R (FEDER Founds) of the Spanish Ministry of Economy and Competitiveness and project BigDaP-TOOLS - Ayudas Fundación BBVA a Equipos de Investigación Científica 2016.

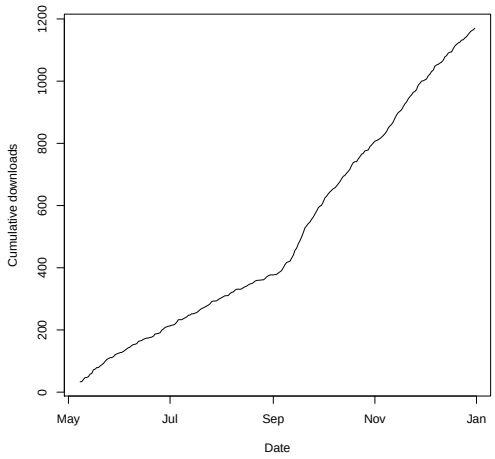


Figure 4: Cumulative downloads since `ruta` was published.

## References

[1] I. T. Jolliffe, Introduction, in: Principal component analysis, Springer, 1986, pp. 1–7. doi:10.1007/978-1-4757-1904-8.

[2] W. S. Torgerson, Multidimensional scaling: I. theory and method, Psychometrika 17 (4) (1952) 401–419. doi:10.1007/BF02288916.

[3] J. B. Tenenbaum, V. De Silva, J. C. Langford, A global geometric framework for nonlinear dimensionality reduction, science 290 (5500) (2000) 2319–2323. doi:10.1126/science.290.5500.2319.

[4] S. T. Roweis, L. K. Saul, Nonlinear dimensionality reduction by locally linear embedding, science 290 (5500) (2000) 2323–2326. doi:10.1126/science.290.5500.2323.

[5] L. v. d. Maaten, G. Hinton, Visualizing data using t-sne, Journal of machine learning research 9 (Nov) (2008) 2579–2605.

[6] I. Goodfellow, Y. Bengio, A. Courville, Deep Learning, MIT Press, 2016, Ch. Deep generative models, pp. 651–716, http://www.deeplearningbook.org.

[7] G. E. Hinton, Reducing the dimensionality of data with neural networks, Science 313 (5786) (2006) 504–507. doi:10.1126/science.1127647.
URL http://www.sciencemag.org/cgi/doi/10.1126/science.1127647

[8] F. Chollet, et al., Keras, https://github.com/fchollet/keras (2015).

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, et al., Tensorflow: A system for large-scale machine learning, in: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, USENIX Association, Berkeley, CA, USA, 2016, pp. 265–283.

[10] D. Charte, F. Charte, S. García, M. J. del Jesus, F. Herrera, A practical tutorial on autoencoders for nonlinear feature fusion: Taxonomy, models, software and guidelines, Information Fusion 44 (2018) 78 – 96. doi:10.1016/j.inffus.2017.12.007.

[11] E. LeDell, N. Gill, S. Aiello, A. Fu, A. Candel, C. Click, T. Kraljevic, T. Nykodym, P. Aboyoun, M. Kurka, M. Malohlava, h2o: R Interface for 'H2O', r package version 3.20.0.2 (2018).
URL https://CRAN.R-project.org/package=h2o

[12] E. Dubossarsky, Y. Tyshetskiy, 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

## Required Metadata

## Current executable software version

Table 1

| Nr. | (executable) Software metadata description | Please fill in this column |
|---|---|---|
| S1 | Current software version | 1.1.0 |
| S2 | Permanent link to executables of this version | `https://github.com/fdavidcl/ruta/releases/tag/1.1.0` |
| S3 | Legal Software License | GPL-3.0 |
| S4 | Computing platform/Operating System | Linux, OS X, Microsoft Windows |
| S5 | Installation requirements & dependencies | Python, R, Tensorflow, Keras |
| S6 | If available, link to user manual - if formally published include a reference to the publication in the reference list | `https://ruta.software/reference/` |
| S7 | Support email for questions | fdavidcl@ugr.es |

Table 1: Software metadata

## Current code version

Table 2

| Nr. | Code metadata description | Please fill in this column |
|---|---|---|
| C1 | Current code version | 1.1.0 |
| C2 | Permanent link to code/repository used of this code version | `https://github.com/fdavidcl/ruta/tree/1.1.0/` |
| C3 | Legal Code License | GPL-3.0 |
| C4 | Code versioning system used | git |
| C5 | Software code languages, tools, and services used | R |
| C6 | Compilation requirements, operating environments & dependencies | R developer tools |
| C7 | If available Link to developer documentation/manual | `https://ruta.software/reference/` |
| C8 | Support email for questions | fdavidcl@ugr.es |

Table 2: Code metadata