A Review of R Neural Network Packages (with NNbenchmark): Accuracy and Ease of Use

by Salsabila Mahdi, Akshaj Verma, Christophe Dutang, Patrice Kiener, John C. Nash

Abstract

In the last three decades, neural-networks have evolved from an academic topic to a common scientific computing tool. CRAN currently hosts around 80 packages (May 2020) that involve neural-network modeling; some offering more than one algorithm. However, to our knowledge, there is no comprehensive study which tests the accuracy, the reliability, and the ease-of-use of those NN packages.

In this paper, we test a large number of packages against a common set of datasets with varying levels of complexity to benchmark and rank them with statistical metrics.

We restrict our evaluation to single hidden-layer perceptrons that perform regression. We ignore packages for classification and other specialized purposes. This leaves us with approximately 60 `package:algorithm` pairs to test. The criteria used in our benchmark were: (i) accuracy, i.e. the ability to find the global minima on 13 datasets, measured by the Root Mean Square Error (RMSE) in a fixed number of iterations; (ii) speed of the training algorithm; (iii) availability of helpful utilities; (iv) quality of the documentation.

We have given a score for each evaluation criterion to compare all `package:algorithm` pairs in a global table. Overall, 15 pairs are considered accurate and reliable and are recommended for daily usage. Other packages are either less accurate, slow, difficult to use, or have poor or zero documentation.

To carry out this work, we developed multiple scripts along with the **NNbenchmark** package. We have open-sourced our code for reproducibility which is available at `https://akshajverma.com/NNbenchmarkWeb/index.html` and **NNbenchmark**.

Introduction

The R Project for Statistical Computing, as any open-source platform, relies on its contributors to keep it up to date. Neural Networks, inspired by the brain itself, are a class of models in the growing field of machine learning for which R has a number of tools. Previously, neural networks were often considered theoretically instead of pragmatically, partly because the algorithms used were computationally expensive.

The term "neural-network" is colloquially used for different model structures and applications. For regression and classification, the term multilayer perceptron is used interchangeably. The term "Recurrent Neural Network" is mainly used in the context of autoregressive time-series while the term "Convolutional Neural Networks" for dimension reduction and pattern recognition (images/audio/text). Most of the above types of neural networks can be found in R packages hosted on CRAN but without any study about the accuracy or the speed of computation. This is an issue as many slow or poor algorithms are available in the literature and hence poor packages are implemented on CRAN.

A neural network algorithm requires complicated calculations to improve the model control parameters. As with other optimization problems, the gradient of the chosen cost function indicates the model's lack of suitability. Optimization methods improve the current iterate by changing the parameters in the opposite of the gradient direction generally with an adaptive step. Parameters for the model are generally obtained by using part of the available data (a training set) and tested on the remaining data. Modern software allows much of this work, including approximation of the gradient, to be carried out without a large effort by the user.

The training process can generally be made more efficient if we can also approximate second derivatives of the cost function, allowing us to use its curvature via the Hessian matrix. There are a large number of approaches, of which quasi-Newton algorithms are perhaps the most common and useful. Within this group, methods based on the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm for updating the (inverse) Hessian approximation provide several well-known examples. In conducting this study, we hypothesize that these second-order algorithms would perform better than the first-order methods for datasets that fit in memory.

To test our hypothesis, we conduct a thorough examination of these training algorithms in R. There are many packages, but there's a dearth of information that would allow the users to make an informed decision. Our work aims to provide a framework for benchmarking neural-network packages. We focus our examination to neural networks of the perceptron type which consist of one input layer, one normalized layer, one hidden layer with a non-linear activation function which is usually the hyperbolic tangent $\tanh()$, and one output layer.

Specifically, we focus only on regression based algorithms. The criteria used in our benchmark were: (i) accuracy, i.e. the ability to find the global minima on 13 datasets, measured by the Root Mean Square Error (RMSE) in a fixed number of iterations; (ii) speed of the training algorithm; (iii) availability of helpful utilities; (iv) quality of the documentation.

Neural Networks: The Perceptron

In this section, we briefly describe the single hidden-layer perceptron. As the "layer" term suggests - some terms come from graphs representations while others come from the traditional literature on non-linear models.

Using the graph description, a single-hidden layer neural network is made up of 3 parts: (i) layer of the input(s), (ii) hidden layer which consists of independent neurons, each of them performing two operations: a linear combination of the inputs plus an offset followed by a non-linear function, (iii) output layer which is a linear combination of the output of the previous layer.

The non-linear function used in the hidden layer must have the following four properties: continuous, differentiable, monotonic, and bounded. The logistic (invlogit), hyperbolic tangent (tanh) and arctangent (atan) functions are the usual candidates. The above description has a simple mathematical equivalence. Let us take two examples.

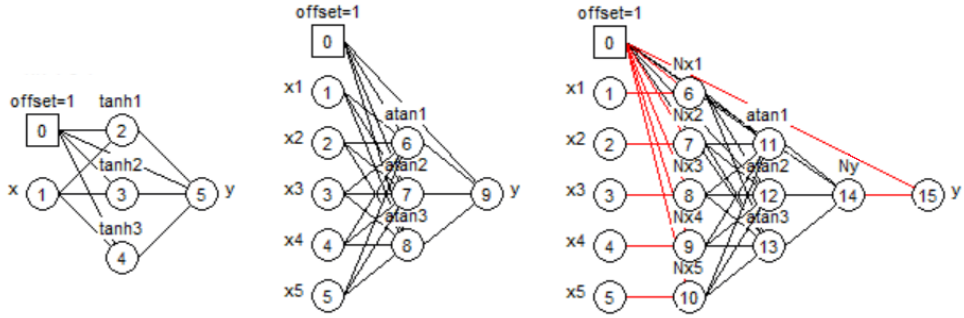
The model $y = a_1 + a_2 \times \tanh(a_3 + a_4 \times x) + a_5 \times \tanh(a_6 + a_7 \times x) + a_8 \times \tanh(a_9 + a_{10} \times x)$ describes a neural network (Fig. 1a) with one input, three hidden neurons, one output model where x is the input, $\tanh()$ is the activation function, y is the output and $a_1, \ldots, a_{10}$ are the parameters.

The model $y = a_1 + a_2 \times \text{atan}(a_3 + a_4 \times x_1 + a_5 \times x_2 + a_6 \times x_3 + a_7 \times x_4 + a_8 \times x_5) + a_9 \times \text{atan}(a_{10} + a_{11} \times x_1 + a_{12} \times x_2 + a_{13} \times x_3 + a_{14} \times x_4 + a_{15} \times x_5) + a_{16}\text{atan}(a_{17} + a_{18} \times x_1 + a_{19} \times x_2 + a_{20} \times x_3 + a_{21} \times x_4 + a_{22} \times x_5)$ describes a neural network (Fig. 1b) with five inputs, three hidden neurons, one output model where x is the input, $\text{atan}()$ is the activation function, y is the output and $a_1, \ldots, a_{22}$ are the parameters.

While the final gradient should be small, we believe it is helpful to have relatively large gradients at the first steps of the training algorithm, so the following is recommended: (i) normalized inputs and outputs (Fig. 1c), (ii) odd functions like the hyperbolic tangent function or the arctangent function, (iii) small random values to initialize the parameters. A common example of this is to use values extracted from a centred Gaussian $\mathcal{N}(0, 0.1)$ distribution.

These practices help us find good local-minima and possibly the global-minima.

The dataset used for the training is assumed to have the number of rows much larger than the number of parameters. While "much larger" is subjective, values of 3 to 5 are



(a) NN 1-3-1 **(b)** NN 5-3-1 **(c)** NN 5-5N-3-1N-1

Figure 1: Three neural networks

generally accepted (in experimental design, some iterative strategies start with a dataset having a number of distinct experiments equal to 1.8 times the number of parameters and then increase the number of experiments to fine tune the model).

It is rather clear from the mathematical formula above that neural networks of perceptron type are non-linear models which require training algorithms that can handle (highly) non-linear models for their parameter estimation. Indeed, the intrinsic and parametric curvatures of such models are usually very high and with so many parameters, the Jacobian matrix might exhibit some co-linearities between its columns and become nearly singular. As a result, appropriate algorithms for such `dataset::model` pairs are rather limited and well-known. They pertain to the class of second-order algorithms such as the BFGS algorithm which is Quasi-Newton in how it updates the approximate inverse Hessian or the Levenberg-Marquardt algorithm which stabilizes the Gauss-Newton search direction at every iteration.

Unfortunately, due to certain educative tools on the backpropagation and recent popularity of "deep neural networks" that manipulate ultra-large models (sometimes more parameters than examples in the datasets), many papers emphasize the use of first-order gradient algorithms.

Therefore, many R packages have implemented such algorithms. In the case of the perceptron, we contend this is an oversight, and provide evidence to that effect in this paper. We refer interested readers to (Tan and Lim, 2019) for a review of second-order algorithms for neural networks.

Methodology

Convergence and termination

Most of the `package:algorithm` pairs try to minimize the Root Mean Squared Error (RMSE) during the training step. Two exceptions are the **brnn** package which minimizes the RMSE plus the sum of the parameters (hence the name Bayesian Regularized Neural Network), and the **qrnn** package which performs quantile regression. For all packages, the datasets were learnt as a whole and without any weighting scheme to favor a single part of a dataset. We don't use a validation/test set because the purpose of our study is to verify the ability to reach good minima. This requirement is satisfied by using only a train set.

When training neural networks, we attempt to tune a set of hyperparameters so that the root to minimize the RMSE. When our method for such adjustment can no longer reduce the RMSE, we say that the given algorithm **terminated**. We consider the method to have **converged** when termination is not due to some exceptional situation and the final RMSE value is relatively small[1]. In practice, some algorithms require that we stop the optimization

[1]We do not choose the mean absolute error (MAE) for overall ranking nor for convergence testing as there is a lack of consensus in the literature, see e.g. (Willmott and Matsuura, 2005; Chai and Draxler, 2014).

process in exceptional situations (e.g., a divide by zero), or a pre-set limit on the number of steps or a maximum elapsed time is reached.

Specifically, second-order algorithms are all set to a maximum of 200 iterations. On the other hand, first-order algorithms were set to several values, depending on how well and how fast they converged: `maxit1storderA=1000` iterations, `maxit1storderB=10000` iterations, and `maxit1storderC=100000` iterations. The full list of the maximum iteration number per `package:algorithm` is given in Table 4 in Appendix D. It can be seen that we were unable to completely harmonize the hyperparameters as an appropriate learning rate differed between packages, despite the algorithm being similarly named.

Performance

We measure **performance** primarily by relative computing time between methods on a particular computing platform. We could count the precise number of iterations, function evaluations or similar quantities that indicate the computing effort, but this would have required a large effort in R coding in order to get values that are comparable between NN packages. We note that differences in machine architecture and in the attached libraries (e.g., BLAS choices for R) will modify our performance measure. We are putting our tools on a Github repository so that further evaluation can be made by ourselves and others as hardware and software evolves.

The majority of the resulting files in our repository were generated on a Windows system build 10.0.18362.752. The machine specifications are - (i) i7-8750H CPU, (ii) Intel(R) UHD Graphics 630, (iii) NVIDIA GeForce GTX 1060 chip, (iv) 16 GB of RAM.

Tests were also performed on other platforms and the computation times were found to be reasonably similar.

Phase 1 - Preparation of benchmark datasets and selection of packages

Datasets

A non-iterative calculation such as Ordinary Least Squares cannot generally be used to model all the datasets in our evaluation set. Varying levels of difficulty in modeling the different data sets are intended to allow us to further classify different algorithms and the packages that implement them. As we focus on regression analysis, we select only datasets where the response variable is real-valued.

Sonja Surjanovic and Derek Bingham of Simon Fraser University created a useful website from which three of the multivariate datasets were drawn. We note the link, name and difficulty level of the three datasets:

- `http://www.sfu.ca/~ssurjano/fried.html`: `mFriedman`, Friedman's dataset, published in (Friedman, 1991) (average difficulty),
- `http://www.sfu.ca/~ssurjano/detpep10curv.html`: `mDette`, Dette's dataset, published in (Dette and Pepelyshev, 2010) (medium difficulty),
- `http://www.sfu.ca/~ssurjano/ishigami.html`: `mIshigami`, Ishigami's dataset, published in (Ishigami and Homma, 1990) (high difficulty).

The last multivariate dataset, `mRef153`, was used to teach neural networks at ESPCI (The City of Paris Industrial Physics and Chemistry Higher Educational Institution, `https://www.neurones.espci.fr/`) from 2003 to 2013 and is available in the proprietary software Neuro One at `http://www.inmodelia.com/software.html`. This dataset presents some interesting non-linear features.

`uDreyfus1` is a pure neural network which has no error. This can make it difficult for algorithms that assume an error exists. `uDreyfus2` is `uDreyfus1` with errors. Both are considered to be of low difficulty and used to teach neural networks at ESPCI from 1991

Table 1: Datasets' summary

Dataset	Row nb.	Input nb.	Neuron nb.	Param. nb.
Multivariate				
mDette	500	3	5	26
mFriedman	500	5	5	36
mIshigami	500	3	10	51
mRef153	153	5	3	22
Univariate				
uDmod1	51	1	6	19
uDmod2	51	1	5	16
uDreyfus1	51	1	3	10
uDreyfus2	51	1	3	10
uGauss1	250	1	5	16
uGauss2	250	1	4	13
uGauss3	250	1	4	13
uNeuroOne	51	1	2	7

to 2013. `uDmod1` and `uDmod2` are univariate datasets with few observations but exhibit high non-linear patterns and prove to be very challenging datasets. The parameters are highly correlated and singular Jacobian matrices often appear.

Three of the univariate datasets were taken from the US National Institute for Standards and Technology (NIST) website: `https://www.itl.nist.gov/div898/strd/nls/nls_main.shtml`. Namely `uGauss1`, `uGauss2` and `uGauss3` published in (Rust, 1996a,b,c, resp.) created by NIST to assess non-linear least squares regressions are of low, low and medium difficulty respectively.

The last univariate dataset, `uNeuroOne`, was also used to teach the same course and is now available in the proprietary software NeuroOne at `http://www.inmodelia.com/software.html`. In Table 1, we list some information on each dataset used in the first round of our analysis: the number of neurons and the induced parameter number are available in the last two columns.

Finally, we consider a Simon Wood test dataset, named `bWoodN1`, used in (Wood, 2011) for benchmarking generalized additive models. Precisely, we consider a generation of Gaussian random variates Y_i, $i = 1, \ldots, n$ with the mean μ_i defined as

$$\mu_i = 1 + f_0(x_{i,0}) + f_1(x_{i,1}) + f_2(x_{i,2}) + f_3(x_{i,3}) + f_4(x_{i,4}) + f_0(x_{i,5})$$

and standard deviation $\sigma = 1/4$ where f_j are Simon Wood's smooth functions defined in Appendix B, $x_{i,j}$ are uniform variates and $n = 20,000$. `bWoodN1` will only be used in the second round of our analysis when the TOP-5 packages will be further analyzed with 5 neurons resulting in 41 parameters.

To build the final result table, we selected all four multivariate datasets and 4 out of the 8 univariate datasets so that the overall score does not overly weight the univariate datasets. Note that the 2020 GSoC results are available in Section 1 of the supplementary materials, (Mahdi et al., 2020). Furthermore the 2019 GSoC code uses all 12 datasets. For convenience, all datasets are made available in **NNbenchmark**, so that anyone can replicate our analysis.

Packages

Using **RWsearch** (Kiener, 2020), we sought to automate the process of searching for neural network packages. All packages that have "neural network" as a keyword in the package title or in the package description were included.

As of May 2020, around 80 packages fall into this category. Packages **nlsr**, **minpack.lm**, **caret** were added because the former two are important implementations of second-order algorithms while the latter is the first cited meta package in the CRAN task view for machine learning, *MachineLearning*. It is also a dependency for some of the other packages tested. Restricting to regression analysis left us with 49 `package:algorithm` pairs in 2019 and 60 `package:algorithm` pairs in 2020.

Phase 2 - Review of packages and development of a benchmarking template

All packages were tested 3 times. Each assessment is described in detail below.

1. The decision to exclude or include

From documentation and example code, we learned that not all packages selected by the automated search fit the scope of our research. Some have no function to generate neural networks while others were not regression neural networks of the perceptron type or were only intended for very specific purposes: for instance to predict the amyloidogenicity propensity of polypeptide sequences. Depending on the package, this could be decided by looking at the `DESCRIPTION` file or by trial and error.

2. Templates for testing accuracy and speed

While inspecting the packages, we slowly developed a template for benchmarking that evolved over time. The final structure of this template (for each package) is as follows:

1. Set up the test environment - loading of packages, setting working directory and options;
2. Summary of tested datasets;
3. Loop over datasets:
    a. setting parameters for a specific dataset,
    b. selecting benchmark options,
    c. training a neural network with a tuned function for each package,
    d. calculation of convergence metrics (RMSE, MAE, WAE)[2],
    e. plot each training over one initial graph, then plot the best result,
    f. add results to the appropriate existing record (*.csv file) and
    g. clear the environment for next loop.
4. Clearing up the environment for the next package. It is optional to print warnings.

To simplify this process, we developed tools in the **NNbenchmark** package, of which the first version was created as part of GSoC'19. In GSoC'20, 3 functions encapsulating the template were added that have been made generic with the extensive use of the `do.call` function from the **base** package:

1. In `trainPredict_1mth1data` a neural network is trained on one dataset and then used for predictions, with several utilities. Then the performance of the neural network is exported, plotted and/or summarized.
2. `trainPredict_1data` serves as a wrapper function for `trainPredict_1mth1data` for multiple methods.
3. `trainPredict_1pkg` serves as a wrapper function for `trainPredict_1mth1data` for multiple datasets.

A function for the summary of accuracy and speed, `NNsummary`, was also added. The package repository is https://github.com/pkR-pkR/NNbenchmark, with package templates in https://github.com/pkR-pkR/NNbenchmarkTemplates.

An example of a call to `trainPredict_1pkg` is given in Appendix C.

3. Ease of use scoring

We define ease-of-use measures to rate NN packages on their user-friendliness. Based on our understanding of what a user may be required to know or do when using a neural network package, we consider: (i) a measure for the availability of appropriate utility functions (ii) a measure for (non-trivial) examples (iii) a sufficient documentation (well-written manual, vignette(s)) (iv) a measure to rate the clarity of the R call to fit a given neural network.

Our ratings are as follows.

[2]We measure the quality of our model by RMSE, but the mean absolute error (MAE) and the worst absolute error (WAE) may help distinguish packages with close RMSE values. See Appendix A for definition of convergence metrics.

1. Utilities in R to deal with NN
    a. a predict function exists = 1 star
    b. scaling capabilities exist = 1 star
2. Sufficient and reliable documentation
    a. the existence of useful and relevant example(s)/vignette(s)
        - clear, with regression = 2 stars
        - unclear, examples use iris or are for classification only = 1 star
        - no examples = 0 stars
    b. input/output is clearly documented, e.g., what values are expected and returned by a function
        - clear input and output = 2 stars
        - only one is clear = 1 star
        - both are not documented = 0 stars
3. User-friendly call to fit a NN
    a. simple one-line call or a single function = 2 stars
    b. multiple-lines call to a single function = 1 star
    c. multiple-lines call to many functions = 0 stars

Hence, to inform users about the usability of packages, the documentation measure ranges from 0 to 4 stars, while the utility and the R call range from 0 to 2 stars.

Phase 3 - Collection of and analysis of results

Results collection

Looping over the datasets using each package template, we collected results in the relevant package directories that rests in the templates repository. A large number of runs were carried out in order to obtain the best result for every package.

Analysis

To rank the speed and quality of convergence, we have devised the following method:

1. The results datasets are loaded into the R environment as one large list. The dataset names, `package:algorithm` names and all 10 run numbers, durations, and RMSE are extracted from that list.
2. For the duration score (DUR), the duration is averaged by dataset. 3 criteria for the RMSE score by dataset are calculated:
    a. The minimum value of RMSE for each `package:algorithm` as a measure of their best performance;
    b. The median value of RMSE for each `package:algorithm` as a measure of their average performance, without the influence of outliers;
    c. The spread of the RMSE values for each package which is measured by the difference between the median and the minimum RMSE (subsequently referred to as RMSE D51).
3. Then, the ranks are calculated for every dataset and the results are merged into one wide dataframe.
    a. The duration rank only depends on the duration;
    b. For minimum RMSE values, ties are decided by duration mean, then the RMSE median;
    c. For median RMSE values, ties are decided by the RMSE minimum, then the duration mean;

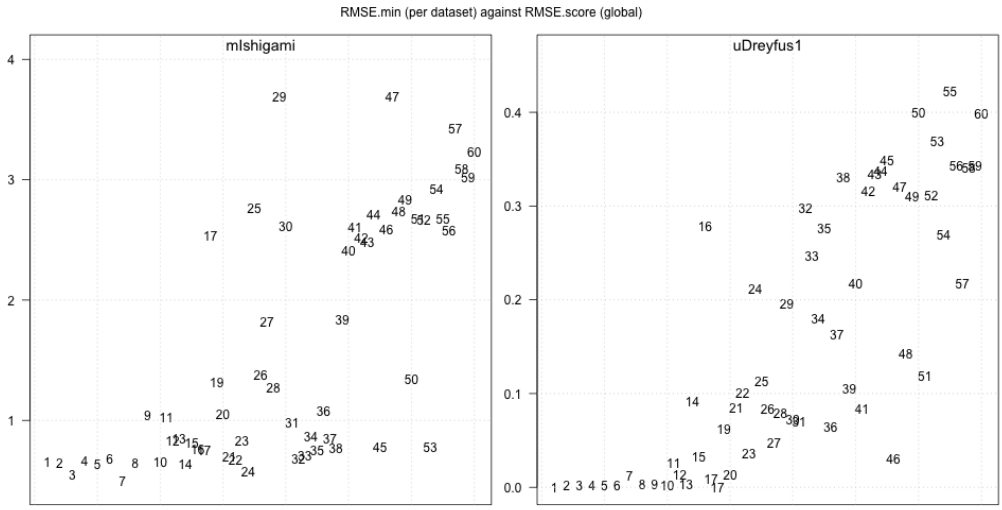


Figure 2: RMSE minimum value per package for `mIshigami` and `uDreyfus1` datasets

d. The RMSE D51 rank only depends on itself.

4. A global score for all datasets is found by a sum of the ranks (of duration, minimum RMSE, median RMSE, RMSE D51) of each `package:algorithm` for each dataset.
5. The final table is the result of ranking by the global minimum RMSE scores for each `package:algorithm`.

Results, discussion and recommendations

Table 2 gives the RMSE and time score per package and per algorithm. The full list of scores is given in Table 4 in Appendix D. We divide our analysis in two groups: packages implementing second-order algorithms and packages implementing first-order algorithms. Figure 2 shows the minimum RMSE value per `package:algorithm` for two particular datasets `mIshigami` and `uDreyfus1`, whereas Figure 3 displays the average computation time. The number on the x-level refers to the RMSE overall score of the `package:algorithm` given in Table 2 (last column), e.g., 8 refers to `validann:optim(CG)` which is a very slow algorithm.

Both figures show that a good overall score does not necessarily imply a good performance on the two datasets under consideration. Furthermore, there is a break between the TOP-10 `package:algorithm` and others in terms of RMSE value. In Section 1.13 of the supplementary materials, (Mahdi et al., 2020), the score probabilities per `package:algorithm` also gives some insights how robust is the overall score.

Regarding computation time, we observe that some `package:algorithm` are very slow and have poor RMSE, e.g. 41 corresponding to `AMORE:BATCHgd`. In the following, we first present the results for second-order algorithms, then low-order algorithms. Finally, we list the reasons for discarded packages.

Second-order algorithms

Of all approaches, the following second-order algorithms generally performed better in terms of convergence despite being limited to $1/5^{th}$ or fewer iterations than the first-order algorithms.

We note that 11 out of 15 of these `package:algorithms` use `optim` from **stats**. 2 of them, **CaDENCE**'s BFGS (Cannon, 2017a) and **validann**'s BFGS and L-BFGS-B (Humphrey, 2017), make the call directly. However, it is not clearly stated in **CaDENCE**'s documentation that `optim`'s BFGS method has been chosen rather than one of the other four methods.

Table 2: Result from Tested Packages

	Individual rating				Global score	
Package	Util	Doc	Call	Algorithm	Time	**RMSE**
nlsr	*	****	**	41. NashLM	18	**1**
rminer	**	***	**	45. nnet_optim(BFGS)	12	**2**
nnet	*	***	**	42. optim (BFGS)	3	**3**
validann	*	****	**	56. optim(BFGS)	35	**4**
	*	****	**	57. optim(CG)	60	**8**
	*	****	**	58. optim(L-BFGS-B)	36	**15**
	*	****	**	59. optim(Nelder-Mead)	55	**45**
	*	****	**	60. optim(SANN)	20	**55**
MachineShop	*	***	*	32. nnet_optim(BFGS)	6	**5**
traineR	*	**	**	55. nnet_optim(BFGS)	4	**6**
radiant.model	**	**	**	44. nnet_optim(BFGS)	10	**7**
monmlp	**	***	**	34. optimx(BFGS)	26	**9**
	**	***	**	35. optimx(Nelder-Mead)	32	**47**
CaDENCE	**	***	**	12. optim(BFGS)	46	**10**
	**	***	**	14. Rprop	56	**51**
	**	***	**	13. pso_psoptim	54	**54**
h2o	**	**		24. first-order	51	**11**
EnsembleBase	*	*	**	23. nnet_optim(BFGS)	5	**12**
caret	**	***	**	15. avNNet_nnet_optim(BFGS)	17	**13**
brnn	**	****	**	11. Gauss-Newton	8	**14**
qrnn	**	***	**	43. nlm()	28	**16**
RSNNS	**	***	**	51. Rprop	24	**17**
	**	***	**	52. SCG	30	**18**
	**	***	**	53. Std_Backpropagation	22	**27**
	**	***	**	47. BackpropChunk	26	**29**
	**	***	**	48. BackpropMomentum	25	**30**
	**	***	**	49. BackpropWeightDecay	29	**31**
	**	***	**	46. BackpropBatch	43	**49**
	**	***	**	50. Quickprop	45	**57**
automl	*	***	**	8. trainwgrad_adam	50	**18**
	*	***	**	9. trainwgrad_RMSprop	47	**26**
	*	***	**	10. trainwpso	57	**43**
deepnet	*	***	**	20. BP	23	**18**
neuralnet	*	***	**	38. rprop+	19	**21**
	*	***	**	37. rprop-	21	**22**
	*	***	**	40. slr	31	**31**
	*	***	**	39. sag	41	**38**
	*	***	**	36. backprop	37	**50**
keras	**	*		28. adamax	48	**23**
	**	*		27. adam	42	**34**
	**	*		29. nadam	44	**36**
	**	*		26. adagrad	58	**37**
	**	*		25. adadelta	59	**40**
	**	*		31. sgd	48	**44**
	**	*		30. rmsprop	37	**52**
AMORE	*	***	*	2. ADAPTgdwm	16	**24**
	*	***	*	1. ADAPTgd	9	**35**
	*	***	*	4. BATCHgdwm	40	**39**
	*	***	*	3. BATCHgd	39	**41**
minpack.lm	*	***	**	33. Levenberg-Marquardt	15	**24**
ANN2	**	***	*	6. rmsprop	14	**28**
	**	***	*	5. adam	13	**33**
	**	***	*	7. sgd	11	**42**
deepdive	**	***	**	16. adam	32	**46**
	**	***	**	19. rmsProp	34	**53**
	**	***	**	18. momentum	53	**56**
	**	***	**	17. gradientDescent	52	**58**
snnR	**	**	**	54. SemiSmoothNewton	7	**48**
elmNNRcpp	**	***	**	21. ELM	1	**59**
ELMR	**	***	**	22. ELM	2	**60**

Note: Statistics over 10 runs.

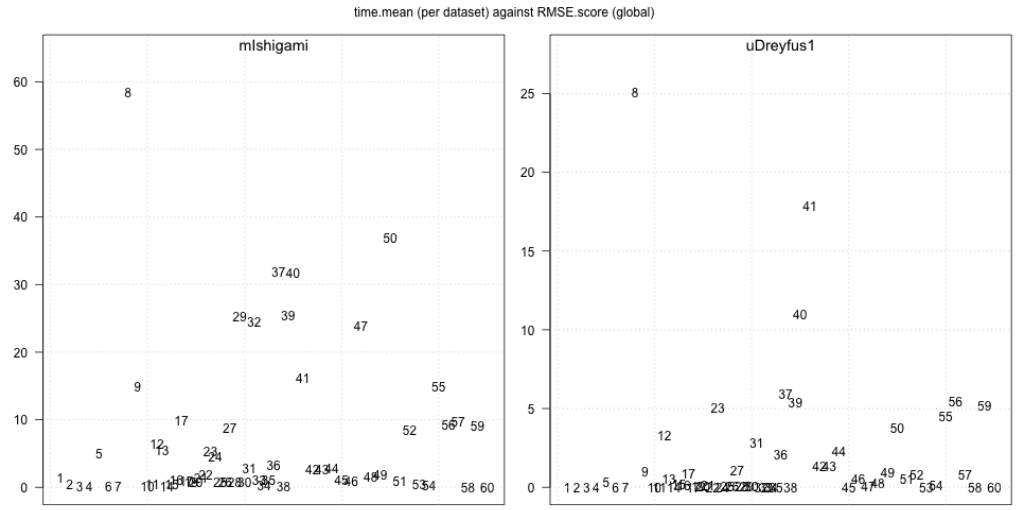


Figure 3: Average time value per package for `mIshigami` and `uDreyfus1` datasets

Furthermore, the mention of Nelder-Mead in the documentation suggests that `optim`'s Nelder-Mead method is used. Speed and variation between results for **CaDENCE** are also not as good as other packages that use `optim`. This could be because **CaDENCE** is intended for probabilistic non-linear models with a full title of "Conditional Density Estimation Network Construction and Evaluation".

By contrast, **validann** is clearly a package that allows a user to use all `optim`'s algorithms. **validann**:`L-BFGS-B` ranks mostly lower than **validann**:`BFGS`, despite the former method being more sophisticated. We believe this is due to our efforts to harmonize parameters, thereby under-utilizing the possibilities of the L-BFGS-B algorithm. Both **CaDENCE** and **validann**'s BFGS are outperformed by **nnet**, especially in terms of speed.

nnet (Ripley, 2020) differs from the two packages above because it uses the C code for BFGS (`vmmin.c`) from `optim` (converted earlier from Pascal) directly instead of calling `optim` from R. This may be what allows it to be faster, but limits the optimization to the single method. **nnet** is only beaten by the Extreme Learning Machine (ELM) algorithms in terms of speed. However, there is a larger variation between results (see the `RMSE D51` in Appendix D) in comparison to **validann**:BFGS. We believe the different default starting values are the cause of this. For instance, **nnet** uses a range of initial random weights of 0.7 while **validann** uses a value of 0.5. In spite of these results, the real reason most authors or users are likely to choose **nnet** is because it is included in the distributed base R and is even mentioned as the very first package in CRAN's task view for machine learning (*MachineLearning*).

Our analysis found that 6 out of 11 packages tested that use `optim` do so through **nnet**. Moreover, approximately 8 packages for neural networks, though not tested, use **nnet**.

The total number of **nnet** dependencies found through a search through the offline database of CRAN with **RWsearch** is 136 packages, although some might be using **nnet** for the multinomial log-linear models, not neural networks.

The packages that use **nnet** for neural networks are often meta packages with a host of other machine learning algorithms. **caret** (Kuhn, 2020), also mentioned in the task-view, boasts 238 methods with 13 different neural network packages, under a deceivingly simple name of "Classification and Regression Training". It has many pre-processing utilities available, as well as other tools.

EnsembleBase (Mahani and Sharabiani, 2016) may be useful for those who wish to make model ensembles and test a grid of parameters, although the documentation is rather confusing. **MachineShop** (Smith, 2020) has 51 algorithms, with some additional information about the response variable types in the second vignette, functions for preprocessing and

tuning, performance assessment, and presentation of results. **radiant.model** (Nijs, 2020) has an unalterable `maxit` of 10000 in the original package. We changed this to harmonize the `maxit` parameter. **rminer** (Cortez, 2020) is the only package dependent on **nnet** that ranks above **nnet** at number 2 for minimum RMSE, and even number 1 in some runs. It also ranks number 1 on the other accuracy measures (median RMSE, minimum MAE, minimum WAE) and is only behind **deepdive** and **minpack.lm** in terms of results that are consistent and do not vary (RMSE D51).

The difference is probably from the change of maximum allowable weights in **rminer** to 10000 from 1000 in **nnet**, which is also probably the reason its fits are slower. **traineR** (Rodriguez R., 2019) claims to unify the different methods of creating models between several learning algorithms.

It is worth noting is that **nnet** and **validann** do not have external normalization, which is especially recommended for **validann** . However, some of the packages dependent on **nnet** do have this capability and it is included in the scoring for ease of use. With **NNbenchmark**, this is done through setting `scale = TRUE` in the function `prepare.ZZ`. Note that use of scaling may complicate the application of constraints, so not be worth the effort for some users. Nevertheless, users might want scaling, or at least to have a clear explanation of the method chosen to center the variables. Scaling of both function and parameters is one of the features that **optimx** (Nash and Varadhan, 2020) incorporates, as some optimization algorithms can work significantly better on scaled problems (Nash, 2014).

Of all the packages, only **monmlp** (Cannon, 2017b) calls **optimx**. Since the calls are for BFGS and Nelder-Mead, they could do better to call `optim` directly, though the door is open to other optimization methods in **optimx**. However, the author, Alex J. Cannon who is also the author of **CaDENCE**, has created a package meant to fill a certain niche, namely for multi-layer perceptrons with optional partial monotonicity constraints. GAM-style effect plots are also an interesting feature. Another package by Cannon is **qrnn** (Cannon, 2019) which uses yet another algorithm: `nlm`, a "Newton-type" algorithm, from **stats**. Although it's performance is at the bottom of second-order algorithms, sometimes even being beaten by first-order algorithms, this could also be because of the intended use of the package compared to the tests here. **qrnn** is designed for quantile regression neural networks, with several options. Cannon has included automatic scaling for all 3 of his packages, as is clearly documented.

stats also includes `nls`, for non-linear least squares, which defaults to an implementation of the second-order algorithm referred to as Gauss-Newton. However, in its documentation, `nls` warns against "zero-residual" or even small residual problems. (Nash, 2014, Section 6.4.1) This was one of the motivations for **nslr** (Nash and Murdoch, 2019). nlsr uses a variant (Nash, 1977) of the Levenberg-Marquardt algorithm versus the plain Gauss-Newton of `nls`, and modifies the relative offset convergence criterion to avoid a zero divide when residuals are small.

minpack.lm (Elzhov et al., 2016) offers another Marquardt approach. Where **nlsr** is entirely in R, and also allows for symbolic or automatic derivatives (which are not relevant to the present study), **minpack.lm** uses compiled Fortran and C code for some important computations. Its structure is also better adapted to use features already available in `nls` that may be important for some uses.

Despite the 2 packages ultimately performing well on all runs (capable of being in the top 3 for RMSE and not slow), there are some reasons why users might hesitate to choose them.

First, both require the full formula of the neural network including variables and parameters. Secondly, they require good starting values to achieve the best convergence. Notice that in Table 2, `minpack.lm` does not have a high rank. This is because we removed the random Gaussian start values we had originally used; which means that the default start values of `minpack.lm` were not appropriate for our datasets. We suspect **nlsr**'s performance on convergence would have similarly dropped if it was possible to use **nlsr** with no user-set starting values and the author's chosen default values were inadequate. `nls` deals with this

by suggesting a companion function in **stats**, `selfStart`. Finally, both packages were able to find better minima when the dataset was scaled. With no starting values and no scaling, `minpack.lm:nlsLM` fails on `uNeuroOne` but performance is better on Friedman & Ishigami datasets. On the other hand, with no start values and no scaling, it fails on everything but `mFriedman`, `mIshigami`, `uDmod2`, and the Dreyfus datasets. Similarly, there is also a notable drop in performance for **nlsr** without scaling on the Gauss datasets and `mRef153`. To conclude, both packages provide algorithms that are capable of doing well on our datasets, but may not be suitable for less experienced users. The vignettes for **nlsr** and earlier book (Nash, 2014) may be useful.

brnn (Rodriguez and Gianola, 2020) is an implementation of the Gauss-Newton algorithm in R that does not rely on `nls` or `nlm` from **stats**. Although it is well-documented and has good speed, **brnn**'s implementation of the Gauss-Newton algorithm still ranks below some of the previously mentioned BFGS and Levenberg-Marquardt tools in terms of its global minimum RMSE. We found 2 reasons that we believe to be the cause of this. First, its model uses one parameter fewer than the other algorithms. Only datasets `uDreyfus1` and `uDreyfus2` which are purely 3 hidden neurons ignore the first term. Second, **brnn** does not minimize the sum of squares of the errors but the sum of squares of the errors plus a penalty on the parameters. In certain circumstances – especially with an almost singular Jacobian matrix as with `mDette`, `mIshigami`, `mRef153`, `uGauss3`, and `uNeuroOne` – this will avoid issues with highly correlated parameters.

The only second-order algorithm which we are unable to recommended from the results of our research is **snnR** (Wang et al., 2017). It ranked among the 10 worst algorithms for minimum RMSE out of all 60 algorithms, but this package, focusing on Sparse Neural Networks for Genomic Selection in Animal Breeding, might prove useful in that perspective.

Lower-order algorithms

Packages with first-order algorithms can be broadly categorized into 2 types: (a) those that allow for one hidden layer (b) those that allow for more than one hidden layer.

A. One hidden layer

The first category is comprised of either packages that also include second-order algorithms previously discussed or packages that use the Extreme Learning Machine algorithm. Only 2 packages include both second-order algorithms and a lower-order algorithm, that is, **monmlp** and **validann**. **monmlp** has one algorithm besides BFGS, that is, **optimx**'s Nelder-Mead. **validann** provides the same algorithm but from `optim`. **validann**'s implementation is slower, as before, but ranks slightly better for minimum RMSE. Both implementations of Nelder-Mead do not rank well in minimum RMSE, around 40 out of 60, with similar ranks for the other criteria. We would also caution users to avoid the other methods in **validann** from `optim`. From Table 2 it may appear that **validann**'s implementation of the Conjugate Gradient (CG) algorithm finds reasonable minima and thus is a good option. It consistently ranked in the top 15 with minimum RMSE. However, it is the slowest algorithm of all 60 algorithms tested. Note, this includes algorithms from packages that call external libraries outside R in Python or Java and packages that use as much as 100,000 iterations.

On the other hand, **validann**'s SANN algorithm is relatively worse than other packages as it ranks at number 55 for minimum RMSE although it is in the top one third for speed (rank 20).

Packages that implement the ELMR algorithm are similar to SANN from **validann** in the sense that they are faster but do not converge as well as other package's algorithms. The 2 packages that do so, **elmNNRcpp** (Mouselimis and Gosso, 2020) and **ELMR** (Petrozziello, 2015) are, respectively, number 1 and number 2 in the ranks for time but 59 and 60 (bottom 2) for minimum RMSE. **ELMR** converges slightly worse on all datasets than **elmNNRcpp** but has noticeably worse performance on the Gauss datasets, especially `uGauss1`. Even increasing the number of neurons did not lead to better convergence for those particular datasets.

B. More than one hidden layer

Following the trend of "deep learning", the last 9 packages provide the option for more than one layer with a first-order learning algorithm. Our results show that they are often either/both slower or worse at converging than the second-order algorithms with the same number of neurons or layers than their counterparts. We recommend choosing better algorithms over more layers for datasets similar to the ones we used.

Choosing more layers often comes at the expense of speed. An example of this is the implementation of the first-order algorithm in **h2o** (LeDell et al., 2020). With the same numbers of neurons it already is quite slow - coming in at 51 out of the 60 algorithms.

With a default hidden layer size of 2, each with 200 neurons, it takes around 10 minutes on `mFriedman` with a minimum RMSE of 0.0022. On the other hand, **nnet** can find a minima of the error function with a minimum RMSE of 0.0088 in less than a second with fewer neurons and only one layer. Thus, despite having a ranking of 11 in minimum RMSE in the final run, beating some of the second-order algorithms, users of **h2o** should be wary of the trade off between performance and speed. Moreover, users might hesitate as it is not actually clear what algorithm is used. The large number of options to choose from seem capable of changing the basic algorithm itself into what is considered a different algorithm by other packages (example: "adaptive_rate: Specify whether to enable the adaptive learning rate (ADADELTA). This option is enabled by default." in link, set to false in latest run). Some users also might not want to setup Java, which is needed, although it is not as painful to setup as some external libraries.

By far, the hardest package to set up which called external libraries was **tensorflow** (Allaire and Tang, 2020) and its derivatives. In the summer of 2019, it took quite some time to figure out how things worked. Then the latest TensorFlow 2.2.0 became available and we hoped to be able to use the Eager Execution provided to avoid the R Session crashing in the summer of 2020. Unfortunately, this led to different problems with the translation between R and Python so we could not use the 2019 code. **tfestimators** (Allaire et al., 2018) also had similar issues and is even less supported. **kerasR** (Arnold, 2017), which provides a consistent interface to Keras, a Python API which provides an easier use interface to TensorFlow, had the same issue. In the end, we tested the algorithms in **keras** (Allaire and Chollet, 2020) with the hope that it would be able to represent the performance of the other packages.

keras has the second-most number of algorithms, a total of 7, with most of them being "adaptive" algorithms. The highest ranking algorithm for minimum RMSE is `adamax` at 23 and the highest ranking algorithm for speed was `rmsprop` at 37 (quite slow). However, these results were achieved with a reasonable GPU so users might want to decide on whether to use **keras** based on their own hardware specifications. Other algorithms did not perform well in terms of minimum RMSE and the spread of RMSE represented by RMSE D51. As **keras** also has many options available, including a convolutional layer for CNNs, more experienced users may prefer it. On the other hand, just deciding the learning rate (the default was not appropriate for our datasets) can be a challenge.

The default learning rates in **RSNNS** (Bergmeir, 2019) were more appropriate to use directly. **RSNNS** is an example of a package that directly wraps around an external library, the Stuttgart Neural Network Simulator (SNNS), to provide an easy-to-use interface. This library is rather large with many implementations of neural networks. It contains the biggest number of algorithms tested at a total of 8. Algorithms `Rprop` and `SCG`, the best for minimum RMSE, rank at 16 and 17 respectively which is pretty good for a first-order algorithm. Speed for `Rprop` is better but `SCG`'s results vary less.

Other packages

AMORE (Limas et al., 2020): Unfortunately, the focus of the paper behind this package, its unique point, is not explained or documented well enough.

An addition of some examples using the TAO option as the error criterion would be helpful for using the TAO-robust learning algorithm, since this type of error measure is most useful for data with outliers. The function for creating a dot file to use with

http://www.graphviz.org is also interesting. ADAPT algorithms appear to perform better than the BATCH algorithms with the parameters used in this research.

ANN2 (Lammers, 2020): This package's implementation of adam or rmsprop consistently ranked in the top half for minimum RMSE which is not bad for a first-order algorithm. It is not as accurate as second-order algorithms but all its algorithms are quite fast. C++ code was used to enhance the speed. Functions for autoencoding are included with anomaly detection in mind.

automl (Boulangé, 2020): It would be easier to use the algorithms in this package if they did not rely on the beta parameters and instead had an argument of their own. However, it is nice that there are notes on what parameters have a higher tuning priority. The package is rather slow (highest ranking algorithm for speed is `RMSprop` at 47) with good enough convergence (highest ranking is `adam` at 18).

deepdive (Balakrishnan, 2020): All algorithms are very good in terms of little variance between results (see its RMSE D51 score). However, the results on convergence by minimum RMSE score are not as good with the worst being `gradientDescent` which ranks 3rd from the bottom. There are few exported functions. The novelty of this package is apparently in the `deeptree` and `deepforest` functions it provides.

deepnet (Rong, 2014): This is one of the better performing implementations of the first-order algorithms back-propagation, ranking at 18 for minimum RMSE. It's also relatively fast, ranking at 23 for speed.

neuralnet (Fritsch et al., 2019): Considering that this is the only package that uses 100000 iterations as its maxit parameter (excluding BNN which is not included in the official ranks), it can be considered as not recommended. Nonetheless, the default algorithm, `rprop+` and the similar `rprop-`, managed to rank 20 and 21 respectively, out of 60 algorithms for minimum RMSE. These two also do not do badly in terms of speed. Following, in order, are `slr`, `sag`, and traditional `backprop` as the worst at rank 48 out of 60 for minimum RMSE. Notes on documentation show that is rather difficult to configure this package, and it should probably not be a dependency for other packages that wish to be more certain of the results. For simple datasets, it is less of an issue.

Untested packages

A certain number of packages have been discarded from this study for at least one of the following reasons:

1. For regression but unsuitable for the scope of our research, coded RE in Table 5.
2. For time series, coded TS in Table 5.
3. For classification, coded CL in Table 5.
4. For specific application purpose, coded AP in Table 5.
5. For tools to complement NN's by other packages, coded UT in Table 5.
6. Not actually neural networks and other reasons, coded XX in Table 5.

The full list of untested packages is given in Table 5 in Appendix D.

Further analysis of TOP-5 packages

We perform a second round of analysis with a larger dataset and a focus on the TOP-5 packages given in Table 2. That is, we consider packages **nlsr**, **rminer**, **nnet**, **validann** with algorithm BFGS and **MachineShop**. We fit the NN packages on Simon Wood's Gaussian dataset, see `bWoodN1` in dataset description, which contains 20,000 rows with 6 inputs valued in [0,1] for a (single) numeric output. Due to the non-linear functions considered, see Appendix B, the link between the output and each explanatory variable is highly non-linear which greatly affects the fitting time. Table 3 gives the metric performance over 20 runs of these TOP-5 five packages on `bWoodN1`.

Table 3: Performance on bWoodN1 dataset

Package	Algorithm	RMSE min	RMSE median	RMSE D51	MAE median	WAE median	Time median
MachineShop	32. nnet_optim	3.547	4.756	1.2100	3.901	16.02	**3.40**
nlsr	41. NashLM	3.548	4.706	1.1570	3.801	16.56	**76.73**
nnet	42. optim	3.550	4.706	1.1560	3.801	16.57	**3.38**
rminer	45. nnet_optim	3.366	3.688	0.3218	2.956	15.43	**11.07**
validann	56. optim	3.360	4.497	1.1370	3.711	15.89	**140.80**

Note: statistics taken over 20 runs; time in seconds.

We observe that the minimum RMSE (over 20 runs) is very similar for all packages, yet **rminer** and **validann** are a little ahead of the others. The metrics median RMSE and RMSE D51 reveal how consistent **rminer**'s results are in comparison to other packages. This is further proved by the other metric norms: WAE and MAE. However, regarding computation time **rminer** is the 3rd slowest with **nlsr** being the 2nd slowest and **validann** being the slowest of all. The best two in terms of speed in this class are **nnet** and **MachineShop**. Nevertheless, these TOP-5 packages performs significantly better than other packages, see Section 2.1 of the supplementary materials, (Mahdi et al., 2020).

Figures in Section 2.2 of the supplementary materials, (Mahdi et al., 2020), provides some insights where a package performs reasonably well with respect to one explanatory variable and where the fit misses the correct behavior of an explanatory variable.

Conclusion and perspective

This paper focuses on benchmarking neural network packages available on CRAN to recommend or advise against some packages. Based on **RWsearch**'s outputs in 2019-2020, we selected 26 appropriate packages to analyze in-depth and discarded the other 63 packages. Using **NNbenchmark**, we ranked 60 `package:algorithm` pairs and are happy to note that most of them converge well enough within a reasonable time. Packages reviewed appear to offer essentially the same methods, and second-order algorithms perform generally better than first-order algorithms.

nnet, the most recommended package of our study, ranked third in terms of minimum RMSE, and is probably the most efficient package. **nnet** is notably used by many other packages, such as **MachineShop** and **rminer** respectively ranked fifth and second. **MachineShop** and **rminer** are also very good challengers in our benchmark, in particular when considering a larger dataset. Other packages in the TOP-5, **nlsr** (the best in terms of RMSE minimum) and **validann** are efficient packages but a little bit slower in our analysis.

However, we are disappointed that many of the packages we reviewed had poor documentation, notably **EnsembleBase** and **keras**. We often found it difficult to discover what default starting values were used for model parameters and/or to understand how to change the hyper-parameters.

As the field of neural networks continues to grow, there will always be more algorithms to validate. For current algorithms in R, our research should be extended to encompass more types of neural networks and their data formats (classifier neural networks, recurrent neural networks, and so on). Different rating schemes and different parameters for package functions can also be tried out.

Our work is available online through `https://akshajverma.com/NNbenchmarkWeb/` and is entirely reproducible thanks to **NNbenchmark**. We hope users and package maintainers find our work useful and will provide any necessary feedback.

Acknowledgements

This work was possible due to the support of the Google Summer of Code initiative for R during years 2019 and 2020. Students Salsabila Mahdi (2019 and 2020) and Akshaj Verma

(2019) are grateful to Google for the financial support.

Bibliography

J. Allaire and F. Chollet. *keras: R Interface to 'Keras'*, 2020. URL https://CRAN.R-project.org/package=keras. R package version 2.3.0.0. [p13]

J. Allaire and Y. Tang. *tensorflow: R Interface to 'TensorFlow'*, 2020. URL https://CRAN.R-project.org/package=tensorflow. R package version 2.2.0. [p13]

J. Allaire, Y. Tang, K. Ushey, and K. Kuo. *tfestimators: Interface to 'TensorFlow' Estimators*, 2018. URL https://CRAN.R-project.org/package=tfestimators. R package version 1.9.1. [p13]

T. Arnold. *kerasR: R Interface to the Keras Deep Learning Library*, 2017. URL https://CRAN.R-project.org/package=kerasR. R package version 0.6.1. [p13]

R. Balakrishnan. *deepdive: Deep Learning for General Purpose*, 2020. URL https://CRAN.R-project.org/package=deepdive. R package version 1.0.1. [p14]

C. Bergmeir. *RSNNS: Neural Networks using the Stuttgart Neural Network Simulator (SNNS)*, 2019. URL https://CRAN.R-project.org/package=RSNNS. R package version 0.4-12. [p13]

A. Boulangé. *automl: Deep Learning with Metaheuristic*, 2020. URL https://CRAN.R-project.org/package=automl. R package version 1.3.2. [p14]

A. J. Cannon. *CaDENCE: Conditional Density Estimation Network Construction and Evaluation*, 2017a. URL https://CRAN.R-project.org/package=CaDENCE. R package version 1.2.5. [p8]

A. J. Cannon. *monmlp: Multi-Layer Perceptron Neural Network with Optional Monotonicity Constraints*, 2017b. URL https://CRAN.R-project.org/package=monmlp. R package version 1.1.5. [p11]

A. J. Cannon. *qrnn: Quantile Regression Neural Network*, 2019. URL https://CRAN.R-project.org/package=qrnn. R package version 2.0.5. [p11]

T. Chai and R. R. Draxler. Root mean square error (RMSE) or mean absolute error (MAE)? – arguments against avoiding RMSE in the literature. *Geoscientific model development*, 7(3): 1247–1250, 2014. URL https://doi.org/10.5194/gmd-7-1247-2014. [p3]

P. Cortez. *rminer: Data Mining Classification and Regression Methods*, 2020. URL https://CRAN.R-project.org/package=rminer. R package version 1.4.5. [p11]

H. Dette and A. Pepelyshev. Generalized latin hypercube design for computer experiments. *Technometrics*, 52(4), 2010. URL https://doi.org/10.1198/TECH.2010.09157. [p4]

T. V. Elzhov, K. M. Mullen, A.-N. Spiess, and B. Bolker. *minpack.lm: R Interface to the Levenberg-Marquardt Nonlinear Least-Squares Algorithm Found in MINPACK, Plus Support for Bounds*, 2016. URL https://CRAN.R-project.org/package=minpack.lm. R package version 1.2-1. [p11]

J. Friedman. Multivariate adaptive regression splines. *The Annals of Statistics*, 19(1):1–67, 1991. URL https://doi.org/10.1214/aos/1176347963. [p4]

S. Fritsch, F. Guenther, and M. N. Wright. *neuralnet: Training of Neural Networks*, 2019. URL https://CRAN.R-project.org/package=neuralnet. R package version 1.44.2. [p14]

G. B. Humphrey. *validann: Validation Tools for Artificial Neural Networks*, 2017. URL https://CRAN.R-project.org/package=validann. R package version 1.2.1. [p8]

T. Ishigami and T. Homma. An importance quantification technique in uncertainty analysis for computer models. In *In Uncertainty Modeling and Analysis, 1990. Proceedings., First International Symposium*, volume 94, pages 742–751, 1990. URL https://doi.org/10.1109/ISUMA.1990.151285. [p4]

P. Kiener. *RWsearch: Lazy Search in R Packages, Task Views, CRAN, the Web. All-in-One Download*, 2020. URL https://CRAN.R-project.org/package=RWsearch. R package version 4.8.0. [p5]

M. Kuhn. *caret: Classification and Regression Training*, 2020. URL https://CRAN.R-project.org/package=caret. R package version 6.0-86. [p10]

B. Lammers. *ANN2: Artificial Neural Networks for Anomaly Detection*, 2020. URL https://CRAN.R-project.org/package=ANN2. R package version 2.3.3. [p14]

E. LeDell, N. Gill, S. Aiello, A. Fu, A. Candel, C. Click, T. Kraljevic, T. Nykodym, P. Aboyoun, M. Kurka, and M. Malohlava. *h2o: R Interface for the 'H2O' Scalable Machine Learning Platform*, 2020. URL https://CRAN.R-project.org/package=h2o. R package version 3.30.0.1. [p13]

M. C. Limas, J. B. O. Mere, A. G. Marcos, F. J. M. de Pison Ascacibar, A. V. P. Espinoza, F. A. Elias, and J. M. P. Ramos. *AMORE: Artificial Neural Network Training and Simulating*, 2020. URL https://CRAN.R-project.org/package=AMORE. R package version 0.2-16. [p13]

A. S. Mahani and M. T. Sharabiani. *EnsembleBase: Extensible Package for Parallel, Batch Training of Base Learners for Ensemble Modeling*, 2016. URL https://CRAN.R-project.org/package=EnsembleBase. R package version 1.0.2. [p10]

S. Mahdi, A. Verma, C. Dutang, P. Kiener, and J. Nash. Supplementary materials for the paper 'a review of R neural network packages (with NNbenchmark): Accuracy and ease of use'. Technical report, Zenodo, 2020. URL https://doi.org/10.5281/zenodo.4360393. [p5, 8, 15]

L. Mouselimis and A. Gosso. *elmNNRcpp: The Extreme Learning Machine Algorithm*, 2020. URL https://CRAN.R-project.org/package=elmNNRcpp. R package version 1.0.2. [p12]

J. C. Nash. Minimizing a nonlinear sum of squares function on a small computer. *Journal of the Institute for Mathematics and its Applications*, 19:231–237, 1977. URL https://doi.org/10.1093/imamat/19.2.231. JNfile: 77IMA J Appl Math-1977-NASH-231-7.pdf. [p11]

J. C. Nash. *Nonlinear Parameter Optimization Using R Tools*. John Wiley & Sons: Chichester, May 2014. ISBN 978-1-118-56928-3. Companion website (see http://www.wiley.com/legacy/wileychi/nash/). [p11, 12]

J. C. Nash and D. Murdoch. *nlsr: Functions for Nonlinear Least Squares Solutions*, 2019. URL https://CRAN.R-project.org/package=nlsr. R package version 2019.9.7. [p11]

J. C. Nash and R. Varadhan. *optimx: Expanded Replacement and Extension of the 'optim' Function*, 2020. URL https://CRAN.R-project.org/package=optimx. R package version 2020-4.2. [p11]

V. Nijs. *radiant.model: Model Menu for Radiant: Business Analytics using R and Shiny*, 2020. URL https://CRAN.R-project.org/package=radiant.model. R package version 1.3.10. [p11]

A. Petrozziello. *ELMR: Extreme Machine Learning (ELM)*, 2015. URL https://CRAN.R-project.org/package=ELMR. R package version 1.0. [p12]

B. Ripley. *nnet: Feed-Forward Neural Networks and Multinomial Log-Linear Models*, 2020. URL https://CRAN.R-project.org/package=nnet. R package version 7.3-14. [p10]

P. P. Rodriguez and D. Gianola. *brnn: Bayesian Regularization for Feed-Forward Neural Networks*, 2020. URL https://CRAN.R-project.org/package=brnn. R package version 0.8. [p12]

O. Rodriguez R. *traineR: Predictive Models Homologator*, 2019. URL https://CRAN.R-project.org/package=traineR. R package version 1.0.0. [p11]

X. Rong. *deepnet: deep learning toolkit in R*, 2014. URL https://CRAN.R-project.org/package=deepnet. R package version 0.2. [p14]

B. Rust. Nonlinear least square regression: Gauss1 dataset. Technical report, NIST, 1996a. URL https://www.itl.nist.gov/div898/strd/nls/data/LINKS/DATA/Gauss1.dat. [p5]

B. Rust. Nonlinear least square regression: Gauss2 dataset. Technical report, NIST, 1996b. URL https://www.itl.nist.gov/div898/strd/nls/data/LINKS/DATA/Gauss2.dat. [p5]

B. Rust. Nonlinear least square regression: Gauss3 dataset. Technical report, NIST, 1996c. URL https://www.itl.nist.gov/div898/strd/nls/data/LINKS/DATA/Gauss3.dat. [p5]

B. J. Smith. *MachineShop: Machine Learning Models and Tools*, 2020. URL https://CRAN.R-project.org/package=MachineShop. R package version 2.5.0. [p10]

H. H. Tan and K. H. Lim. Review of second-order optimization techniques in artificial neural networks backpropagation. In *IOP Conference Series: Materials Science and Engineering*, volume 495, page 012003. IOP Publishing, 2019. URL https://doi.org/10.1088/1757-899X/495/1/012003. [p3]

Y. Wang, P. Lin, Z. Chen, Z. Bao, and G. J. M. Rosa. *snnR: Sparse Neural Networks for Genomic Selection in Animal Breeding*, 2017. URL https://CRAN.R-project.org/package=snnR. R package version 1.0. [p12]

C. J. Willmott and K. Matsuura. Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate research*, 30(1): 79–82, 2005. URL https://doi.org/10.3354/cr030079. [p3]

S. N. Wood. Fast stable restricted maximum likelihood and marginal likelihood estimation of semiparametric generalized linear models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(1):3–36, 2011. URL https://doi.org/10.1111/j.1467-9868.2010.00749.x. [p5]

Appendix

Appendix A

Consider a set of observations y_i and its corresponding predictions $\hat{y}_i$ for $i = 1, \ldots, n$. The three metrics used were:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|,\ RMSE = \frac{1}{n}\sqrt{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2},\ WAE = \frac{1}{n}\max_{i=1,\ldots,n}|y_i - \hat{y}_i|.$$

These values represent the absolute, the squared and the maximum norm of residual vectors.

Appendix B

We define five smooth functions for Simon Wood's test dataset

$$f_0 = 5\sin(2\pi x),\ f_1 = \exp(3x) - 7,$$

$$f_2 = 0.5 \times x^{11}(10(1-x))^6 - 10(10x)^3(1-x)^{10},\ f_3 = 15\exp(-5|x - 1/2|) - 6,$$

$$f_4 = 2 - 1_{(x<=1/3)}(6x)^3 - 1_{(x>=2/3)}(6-6x)^3 - 1_{(2/3>x>1/3)}(8 + 2\sin(9(x-1/3)\pi)).$$

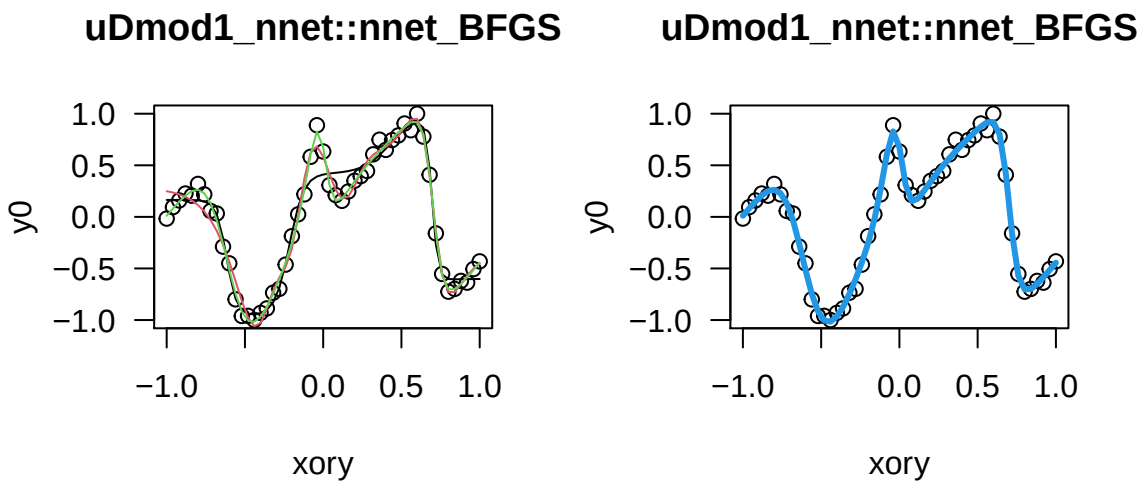


Figure 4: Example of nnet on uDmod1

Appendix C

An example of our template for the package nnet:

```
library(NNbenchmark)
nrep <- 3
odir <- tempdir()

library(nnet)
nnet.method <- "BFGS"
hyperParams.nnet <- function(...) {
    return (list(iter=200, trace=FALSE))
}
NNtrain.nnet <- function(x, y, dataxy, formula, neur, method, hyperParams, ...) {

    hyper_params <- do.call(hyperParams, list(...))

    NNreg <- nnet::nnet(x, y, size = neur, linout = TRUE,
                        maxit = hyper_params$iter, trace=hyper_params$trace)
    return(NNreg)
}
NNpredict.nnet  <- function(object, x, ...) { predict(object, newdata=x) }
NNclose.nnet    <- function() {  if("package:nnet" %in% search())
                                 detach("package:nnet", unload=TRUE) }
nnet.prepareZZ  <- list(xdmv = "d", ydmv = "v", zdm = "d", scale = TRUE)

res <- trainPredict_1pkg(5, pkgname = "nnet", pkgfun = "nnet", nnet.method,
  prepareZZ.arg = nnet.prepareZZ, nrep = nrep, doplot = TRUE,
  csvfile = FALSE, rdafile = FALSE, odir = odir, echo = FALSE)
```

Appendix D

Table 5: Review of Discarded Packages

Package	Category	Reason to Discard (File(s) and/or function(s))
appnn	AP	Provide a feed forward neural network to predict the amyloidogenicity propensity of polypeptide sequences (DESCRIPTION file)

Table 5: Review of Discarded Packages *(continued)*

Package	Category	Reason to Discard (File(s) and/or function(s))
autoencoder	AP	Provide a sparse autoencoder, an unsupervised algorithm that learns useful features from the data its given (::autoencode)
BNN	RE*	Use a feed forward neural network to perform regression. It is unclear whether it fits the form of perceptron in the scope. It states that it is intended for variable selection, although how exactly the package would be used to do so is missing. Also the source code is written in C that users of R might not understand. Performance is slowÂ : need 100.000 iterations. (::BNNsel-examples & abstract of paper)
Buddle	CL	Did not include regression in 2019. Unfortunately, the version we tested in 2020 could not be used properly for regression either. See the examples (::TrainBuddle)
cld2	XX	Provide bindings to Google's C++ library CLD2, which detects languages using a NaÃ¯ve Bayesian classifier. CLD3, which does use neural networks, is mentioned in the description (DESCRIPTION file & link to github)
cld3	AP	Bindings to Google's C++ library CLD3, which detects languages using a neural network with an experimental algorithm (DESCRIPTION file)
condmixt	AP	Use neural networks to predict parameters of mixture models (DESCRIPTION file)
DamiaNN	RE	Was designed specificly for training datasets from Numerai, <https://numer.ai/>. We were unable to adapt it to our datasets even after exporting functions from the interactive interface (DESCRIPTION file, help pages)
deep	CL	Seem to implement a perceptron to classify data (implicitly known from choice of iris as example and in source code)
deepNN	RE	Another implementation of deep learning. Its input format of lists of vectors is not standard require users to understand how to use lapply or other functions to convert the format of their data. Univariate datasets can't be used with the functions and we could not manage to adapt it to 2020 code (::train).
DNMF	XX	Help extract features that enforce spatial locality with separability between classes in a discriminant manner (DESCRIPTION file)
evclass	CL	Provide an evidential neural network that outputs Dempster-Shafer mass functions (DESCRIPTION file)
gamlss.add	UT	Allow users to use nnet with a variety of Generalized Additive Models for Location Scale and Shape (::nn). It is not particularly appropriate for all our datasets.
gcForest	XX	Based on an article with "Towards an Alternative to Deep Neural Networks" in its title (DESCRIPTION file)
GMDH	TS	Provide GMDH type neural network algorithms for short term forecasting on a univariate time series (DESCRIPTION file)
GMDH2	CL	Provide GMDH type neural network algorithms for performing binary classification (DESCRIPTION file)
GMDHreg	RE*	Regression using GMDH algorithms. We only managed to tested the COMBI algorithm (the most basic and first in the vignette) on the multivariate datasets. It is strangely slow on the "easy" datasets, mFriedman and mRef153. The convergence is relatively not good considering the ammount of layers (Title in DESCRIPTION file)
gnn	AP	Out of scope: Generative moment matching networks (GMMNs) are introduced for generating quasi-random samples from multivariate models (article abstract)
grnn	RE	Provide an implementation of Specht's General Regression Neural Network in 1991 (DESCRIPTION file). We could not manage to make the functions work on the multivariate datasets. ::guess, the function for predicting, only allows for 1 data at a time. Performance of General Regression Neural Networks can be seen from package yager instead.
hybridEnsemble	RE	Hybrid ensemble of eight different sub-ensembles (DESCRIPTION file)
image.libfacedetection	AP	Face detection with CNNs (DESCRIPTION file)
isingLenzMC	AP	Out of scope: This package provides utilities to simulate one dimensional Ising Model with Metropolis and Glauber Monte Carlo (DESCRIPTION file)
kerasR	RE	See section on keras
leabRa	RE	Provide the local error driven and associative biologically realistic algorithm (Leabra) from O'Reilly 1996. It combines supervised and unsupervised learning, so out of scope (DESCRIPTION file).
learNN	CL	Implement some basic neural networks from \url{http://qua.st/} (DESCRIPTION file). Examples seem to focus on binary classification (::learn_gd, ::learn_bp).
LilRhino	AP	Provide binary neural networks meant for reducing data (DESCRIPTION file), a random forest style collection of neural networks for classification (::Random_Brains), and code for even more purposes. Documentation is satisfyingly clear for a package for applications: a 3 layer network with an adam optimizer, with an explanation of its activation functions (::Binary_Network)
neural	CL	An implementation of "a simple MLP neural network that is suitable for classification tasks" (::mlptrain)
NeuralNetTools	UT	Out of scope: Functions are available for plotting, quantifying variable importance, conducting a sensitivity analysis, and obtaining a simple list of model weights (DESCRIPTION file and Help Pages titles)
NeuralSens	UT	A greater focus on sensitivity, with additional functions (DESCRIPTION file)

Table 5: Review of Discarded Packages *(continued)*

Package	Category	Reason to Discard (File(s) and/or function(s))
NlinTS	TS	A non-linear version of a causality test with feed forward neural networks and a Vector Auto-Regressive Neural Network (VARNN) for non-linear time series analysis models (DESCRIPTION file)
nnetpredint	UT	Out of scope: Computing prediction intervals of neural network models at certain confidence level (DESCRIPTION file)
nnfor	TS	Automatic to fully manual time series modelling with neural networks (DESCRIPTION file)
nnlib2Rcpp	CL	Provide a collection of neural networks, but examples seem to indicate classification and testing our code with the functions provided led to error. Using the RcppClass might be confusing for less experienced R users (::NN-class)
nntrf	AP	Provide useful pre-processing for Machine Learning tasks through data transformation in a non-linear, supervised way with a perceptron (DESCRIPTION file)
onnx	UT	Aims to provide an open source format for neural networks, with definitions of an extensible computation graph model, built-in operators, and standard data types (DESCRIPTION file)
OptimClassifier	UT	Search for the best amount of neurons for binary classifcation neural networks, among other types of binary classifiers (based on how Optim.NN works & DESCRIPTION file)
OSTSC	UT	A tool to solve imbalanced data for univariate time series classification with oversampling using integrated ESPO and ADASYN methods (DESCRIPTION file) thus improving the performance of RNN classifiers (vignette)
passt	AP	This package provides implementation of the Probability Associator Time (PASS-T) model, a memory model based on a simple competitive artificial neural network which imitates human judgment of frequency and duration (DESCRIPTION file)
pnn	CL	This package provides implementation of the Specht algorithm, 1990, for classification with four functions: learn, smooth, perf, and guess (DESCRIPTION file)
polyreg	XX	Polyregression as alternative to NN (DESCRIPTION file)
predictoR	RE	A shiny interface for supervised learning with very minimal documentation. Users may be additionally confused when opening the application only to find that it's default language is Espanol, although this can be changed in the Idioma section. (DESCRIPTION file & ::init_predictor)
ProcData	AP	Provide tools for exploratory process data analysis via functions: reading, process manipulation, action sequence generators, feature extraction and prediction (link + DESCRIPTION file)
quarrint	AP	Out of scope: provide two indexes for interaction prediction between groundwater and quarry extension, one of which is an artificial neural networkÂ ; specified classifier for quarry data (help page - quarrint-package and DESCRIPTION file)
rasclass	CL	Provide neural networks as one of the five supervised classification algorithms for raster images with a design meant to facilitate land-cover analysis (DESCRIPTION file)
rcane	RE	Provide parameter estimation for linear regression, which was not appropriate for the relationships in our data. (DESCRIPTION file)
regressoR	RE	A manual rich version of predictoR
rnn	AP	Implementations of the vanilla Recurrent Neural Network, Long Short-Term Memory (LSTM), and Gated Recurrent Unit (GRU) in native R (DESCRIPTION file)
RTextTools	AP	Out of scope: A machine learning package for automatic text classification (DESCRIPTION file)
ruta	AP	unsupervised neural networks (DESCRIPTION file)
simpleNeural	CL	Neural networks for multi-class or binary classification (DESCRIPTION file)
softmaxreg	CL	Out of scope: Implementation of 'softmax' regression and classification models with multiple layer neural network (DESCRIPTION file)
Sojourn.Data	AP	Stores some neural networks used for Sojourn Accelerometer methods (DESCRIPTION file)
spnn	CL	Out of scopeÂ : Scale invariant version of the original PNN with the added functionality of allowing for smoothing along multiple dimensions while accounting for covariances within the data set (DESCRIPTION file)
studyStrap	AP	Implements multi-study learning algorithms such as merging, the study-specific ensemble the study strap, the covariate-matched study strap, covariate-profile similarity weighting, and stacking weights with single-study learners from caret (DESCRIPTION file)
TeachNet	CL	Provide neural networks with up to 2 hidden layers, 2 different error functions, and a weight decay for 2 class classificationÂ : it is slow. (DESCRIPTION file & ::TeachNet)
tensorflow	RE	See section on keras
tfestimators	RE	See section on keras
trackdem	AP	An artificial neural network can be trained for filtering false positives present in video materials or image sequences (DESCRIPTION file)
TrafficBDE	RE*	Use caret for a grid of parameters for 3 layers combined with neuralnet. Is very slow. Out of scope to test one layer perceptrons. We recommend the author to use other packages and lessen the number of layers. Datasets in Traffic Status Prediction and Urban Places are similar in nature to ours (TrainCR.R, DESCRIPTION file)

Table 5: Review of Discarded Packages *(continued)*

Package	Category	Reason to Discard (File(s) and/or function(s))
tsfgrnn	TS	Out of scope: A general regression neural network (GRNN) is a variant of a Radial Basis Function Network. Allow you to forecast time series using an autoregressive GRNN model (DESCRIPTION file)
yager	RE*	This package provides a neural network that behaves differently from a perceptron. Results indicate that predictions are quite close to the real values, however this comes at the cost of a large number of weights. With less weights or insufficient training data, the performance isn't as great. (::grnn.fit)
yap	CL	Yet another PNN, with a N-level response, where N > 2 (DESCRIPTION file)
zFactor	AP	Computational algorithms to solve equations and find the 'compressibility' factor 'z' of hydrocarbon gases (DESCRIPTION file)

Note: AP=Application, CL=Classification, RE=Regression, RE*=?, TS=Time serie, UT=Utility, XX=Other.

Salsabila Mahdi
Universitas Syiah Kuala
JL. Syech Abdurrauf No.3, Aceh 23111, Indonesia
bila.mahdi@mhs.unsyiah.ac.id

Akshaj Verma
Manipal Institute of Technology
Manipal, Karnataka, 576104, India
akshajverma7@gmail.com

Christophe Dutang
Université Paris-Dauphine, University PSL, CNRS, CEREMADE
Place du Maréchal de Lattre de Tassigny, 75016 Paris, France
dutang@ceremade.dauphine.fr

Patrice Kiener
InModelia
5 rue Malebranche, 75005 Paris, France
patrice.kiener@inmodelia.com

John C. Nash
Telfer School of Management, University of Ottawa
55 Laurier Avenue East, Ottawa, Ontario K1N 6N5 Canada
nashjc@uottawa.ca

Table 4: All convergence scores per package:algorithm sorted by minimum RMSE

		Input parameter			RMSE Score		Other score	
Package	Algorithm	Input format	Maxit	Learn. rate	median	D51	MAE	WAE
nlsr	41. NashLM	full fmla & data	200	-	3	16	3	6
rminer	45. nnet_optim(BFGS)	fmla & data	200	-	1	6	1	1
nnet	42. optim (BFGS)	x & y	200	-	2	17	2	3
validann	56. optim(BFGS)	x & y	200	-	4	10	4	5
	57. optim(CG)	x & y	1000	-	6	10	5	4
	58. optim(L-BFGS-B)	x & y	200	-	13	30	14	13
	59. optim(Nelder-Mead)	x & y	10000	-	44	45	46	42
	60. optim(SANN)	x & y	1000	-	53	51	56	55
MachineShop	32. nnet_optim(BFGS)	fmla & data	200	-	9	22	9	7
traineR	55. nnet_optim(BFGS)	fmla & data	200	-	5	15	6	2
radiant.model	44. nnet_optim(BFGS)	"y" & data	200	-	8	32	12	10
monmlp	34. optimx(BFGS)	x & y	200	-	10	18	9	11
	35. optimx(Nelder-Mead)	x & y	10000	-	47	45	44	47
CaDENCE	12. optim(BFGS)	x & y	200	-	28	48	21	40
	14. Rprop	x & y	1000	0.01	54	60	52	58
	13. pso_psoptim	x & y	1000	-	56	56	54	56
h2o	24. first-order	"y" & data	10000	0.01	7	7	8	8
EnsembleBase	23. nnet_optim(BFGS)	x & y	200	-	15	34	15	15
caret	15. avNNet_nnet_optim(BFGS)	x & y	200	-	10	21	11	9
brnn	11. Gauss-Newton	x & y	200	-	12	9	13	12
qrnn	43. nlm()	x & y	200	-	14	25	7	36
RSNNS	51. Rprop	x & y	1000	-	23	52	25	28
	52. SCG	x & y	1000	-	17	26	18	19
	53. Std_Backpropagation	x & y	1000	0.1	32	31	31	36
	47. BackpropChunk	x & y	1000	-	34	41	32	34
	48. BackpropMomentum	x & y	1000	-	35	39	35	30
	49. BackpropWeightDecay	x & y	1000	-	30	43	33	31
	46. BackpropBatch	x & y	10000	0.1	48	27	50	48
	50. Quickprop	x & y	10000	-	58	36	58	57
automl	8. trainwgrad_adam	x & y	1000	0.01	20	35	16	20
	9. trainwgrad_RMSprop	x & y	1000	0.01	31	50	29	39
	10. trainwpso	x & y	1000	-	41	49	41	38
deepnet	20. BP	x & y	1000	0.8	18	38	24	17
neuralnet	38. rprop+	fmla & data	100000	-	23	40	23	24
	37. rprop-	fmla & data	100000	-	21	42	21	18
	40. slr	fmla & data	100000	-	39	37	39	46
	39. sag	fmla & data	100000	-	49	59	47	52
	36. backprop	fmla & data	100000	0.001	51	10	49	45
keras	28. adamax	x & y	10000	0.1	18	20	20	16
	27. adam	x & y	10000	0.1	28	44	30	25
	29. nadam	x & y	10000	0.1	39	58	40	41
	26. adagrad	x & y	10000	0.1	43	53	42	35
	25. adadelta	x & y	10000	0.1	35	19	34	33
	31. sgd	x & y	10000	0.1	45	47	45	43
	30. rmsprop	x & y	10000	0.1	55	57	55	54
AMORE	2. ADAPTgdwm	x & y	1000	0.01	22	29	16	26
	1. ADAPTgd	x & y	1000	0.01	25	8	26	21
	4. BATCHgdwm	x & y	10000	0.1	33	14	37	27
	3. BATCHgd	x & y	10000	0.1	38	24	42	31
minpack.lm	33. Levenberg-Marquardt	full fmla & data	200	-	16	5	19	14
ANN2	6. rmsprop	x & y	1000	0.01	25	33	27	23
	5. adam	x & y	1000	0.01	27	27	28	21
	7. sgd	x & y	1000	0.01	37	22	36	29
deepdive	16. adam	x & y	10000	0.4	42	1	38	44
	19. rmsProp	x & y	1000	0.8	46	4	48	50
	18. momentum	x & y	1000	0.8	52	3	53	51
	17. gradientDescent	x & y	10000	0.8	57	2	57	53
snnR	54. SemiSmoothNewton	x & y	200	-	49	13	50	48
elmNNRcpp	21. ELM	x & y	-	-	59	55	59	59
ELMR	22. ELM	fmla & data	-	-	60	53	60	60

Note: TOP5 are nlsr:NashLM, rminer:nnet_optim(BFGS), nnet:optim (BFGS), validann:optim(BFGS), MachineShop:nnet_optim(BFGS).