Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

Random weight neural networks in R: The RWNN package

Søren B. Vilsen
Department of Mathematical Sciences, Aalborg University

Abstract

This paper serves as an introduction to the **RWNN** package. The **RWNN** package implements random weight neural networks. The methods are implemented using a combination of R and C++ offsetting the heavier computation and estimation to C++ through the **Rcpp** and **RcppArmadillo** packages. While implementations of random weight neural networks exist other R packages, these focus on the simplest possible variant of the random weight neural network and cover only very specialised use cases of these networks. Besides a general purpose implementation of random weight neural networks, the **RWNN** package also includes common variants such as deep RWNN and sparse RWNN, as well as ensemble methods using random weight neural networks as the base learner. Furhtermore, the **RWNN** packages also includes more sophisticated methods for initialising the random weights, as well as methods for pruning both the number of weights and the number of neurons in the network.

Keywords: random weight neural networks, regularisation, ensemble learning, Bayesian computation, R, **Rcpp**, **RcppArmadillo**.

1. Introduction

Neural networks, and variants thereof, have seen a massive increase in popularity in recent years. This has largely been due to the flexibility of the neural network architecture, and their accuracy when applied to highly non-linear problems. However, due to the highly non-linear nature of the neural network architecture, estimating the weights of these networks using gradient based optimisation (i.e. back-propagation), can be slow and does not guarantee a globally optimal solution. In order to combat these problems, various simplifications of the feed forward neural network (FFNN) architecture have been proposed, including random weight neural networks (RWNNs). They were first introduced in the early 1990's under the name random vector functional links (RVFL) (Schmidt, Kraaijveld, and Duin 1992, Pao, Park, and Sobajic (1992)), and a simplified version was re-discovered under the name extreme learning machines (ELM) (Huang, Zhu, and Siew 2006) in the mid 2000's. The general idea of RWNNs is to keep the randomly assigned weights of the network between the input-layer and the last hidden-layer fixed, and focus on estimating the weights between the last hidden-layer and the output-layer. In the case of regression, this simplification makes estimating the output weights of a RWNN equivalent to estimating the weights of a (regularised) linear model. Theoretically RWNNs show similar universal approximation properties as their FFNN counterparts, i.e. as the number of neurons tends towards infinity the RWNN should be able to approximate any function arbitrarily well (placing only loose assumptions on the activation function). However, practically the number of neurons needed for this approximation to be acceptable may not be feasible for a particular application. Therefore, extensions of RWNNs have been proposed limiting the number of neurons in favour of deeper architecture, as seen in deep RWNN (Henríquez and Ruz 2018) and ensemble deep RWNN (Shi, Katuwal, Suganthan, and Tanveer 2021), or in favour of sparse unsupervised pre-training of the weights, like sparse RWNN (Zhang, Wu, Cai, Du, and Yu 2019).

Implementations of RWNNs already exist in R through the packages **nnfor** (Kourentzes 2019) and **elmNNRcpp** (Mouselimis 2022), both focusing on the simpler ELM architecture. Both implementations allow for a varying number of neurons in the hidden-layer, as well as the specification of the activation function, but are limited to a single hidden-layer with no functional link between the input- and output-layers. The **nnfor** package was designed for using ELMs to handle time-series data and allows for forecasting at different temporal frequencies using the **thief** package. Furthermore, it allows for estimation of the output-weights using Moore-Penrose inversion, ℓ_1-regularisation, and ℓ_2-regularisation. The **elmNNRcpp** package is the successor to **elmNN** package (Gosso 2012) re-implemented in C++ through **Rcpp** and **RcppArmadillo** (Eddelbuettel and François 2011, Eddelbuettel and Sanderson (2014)). The package is a standard implementation of the ELM architecture for both regression and classification. The package allows the user to specify the leakage of the implemented relu activation function, as well as the tolerance of the Moore-Penrose inversion used to estimate the output-weights.

RWNN is a general purpose implementation of RWNNs in R (R Core Team 2021) focusing on both regression and classification problems. The **RWNN** package allows the user to create an RWNN of any depth, set the number of neurons and activation functions in each layer, choose the sampling distribution of the randomly assigned weights, and choose whether the output weights should be estimated by either Moore-Penrose inversion, ℓ_1-regularisation, or ℓ_2-regularisation. RWNN is implemented in C++ through **Rcpp** and **RcppArmadillo**; along with the standard RWNN implementation the following variants have also been included:

- **ELM** (extreme learning machine) (Huang *et al.* 2006): A simplified version of an RWNN without a link between the input and output layer (this is also the default behaviour of the RWNN implementation).
- **deep RWNN** (Henríquez and Ruz 2018), (Shi *et al.* 2021): An RWNN with multiple hidden layers, where the output of each hidden-layer is included as features in the model.
- **sparse RWNN** (Zhang *et al.* 2019): Applies sparse auto-encoder (ℓ_1 regularised) pre-training to reduce the number non-zero weights between the input and the hidden layer (the implementation generalises this concept to allow for both ℓ_1 and ℓ_2 regularisation).
- **ensemble deep RWNN** (Shi *et al.* 2021): An ensemble extension of deep RWNNs

using the output of each hidden layer to create an ensemble of RWNNs, each hidden-layer being used to create a seperate prediction of the target.

Furthermore, the **RWNN** package also includes general implementations of the following ensemble methods (using RWNNs as base learners):

- **Stacking** (Wolpert 1992), (Breiman 1996b): Stack multiple randomly generated RWNN's, deep RWNN's, or sparse RWNN's and estimate their contribution to a weighted ensemble using k-fold cross-validation.
- **Bagging** (Breiman 1996a), (Breiman 2001), (Sui, He, Vilsen, Teodorescu, and Stroe 2021): Bootstrap aggregation of RWNN's, deep RWNN's, or sparse RWNN's creates a number of bootstrap samples, sampled with replacement from the training-set. Furthermore, as in random forest, instead of using all features when training each RWNN, a subset of the features are chosen at random.
- **Boosting** (Friedman 2001): The boosting implementation is based on residual boosting using RWNN's, deep RWNN's, or sparse RWNN's as the base learner.

As RWNN's will inevitably include a lot of randomly generated features, to improve their computational and memory efficiency, the **RWNN** package also includes methods for pruning the number of weights and neurons using magnitude, mutual information, and Fisher information.

Lastly, the **RWNN** package also includes a simple method for grid based hyperparameter optimisation, where k-fold cross-validation is applied to the training-set to find the hyperparameters yielding the smallest cross-validation error on a user defined grid (similar to the **tune** function found in the package **e1071** (Meyer, Dimitriadou, Hornik, Weingessel, and Leisch 2023)).

The remainder of the paper is structured as follows: Section 2 introduces the general idea of RWNNs, this is followed by the method of estimating the output weights, and an outline of each of the implemented RWNN variants. The utility of the **RWNN** package is shown in Section 3 applying different RWNN variants to a series of examples. Lastly, a conclusion is found in Section 4.

2. Random Weight Neural Networks

RWNN's are simplifications of FFNN's, where the weights between the input-layer and the hidden-layers are kept fixed after being randomly initialised, i.e. only the weights between the last hidden-layer and the output-layer are estimated during the training process. Furthermore, a direct link (also called a functional link) between the features and the output may be included; when the link is active, the RWNN can be seen as a concatenation of the original features, and a random non-linear transformation of these features. Using this simplification, assuming the loss is the squared loss, and that the activation function is linear (i.e. the identity function), then estimation of the output-weights simplifies greatly, as it becomes equivalent to estimating the weights in (regularised) multiple linear regression. The general structure of RWNN's with a single hidden-layer can be seen in Figure 1, where the dashed red lines indicate the functional link.

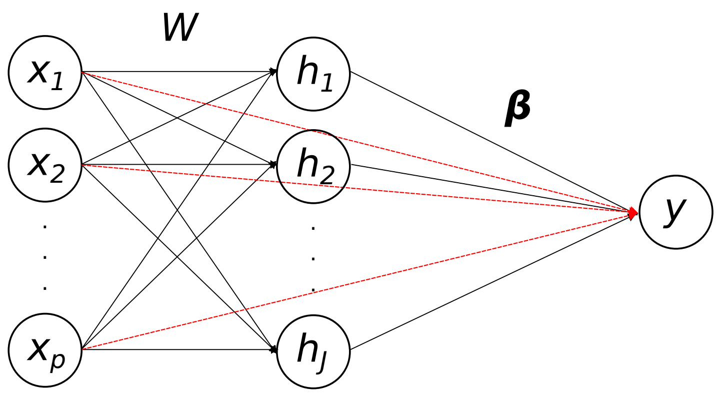

Figure 1: Graph representation of a random weight neural network (RWNN) with funcitonal link between the input and the output layer.

2.1. RWNN

Given a sample of N observations, $\mathcal{D} = \{(\boldsymbol{x}_n, y_n)\}_{n=1}^N$, where $\boldsymbol{x}_n$ is p dimensional vector of features and y_n is the target, of observation n, the output of the j'th neuron in the hidden layer is:

$$h_{nj} = f\left(\sum_{i=1}^{p} w_{ij} x_{ni} + w_0\right), \tag{1}$$

where f is the activation function, w_0 is the bias, and w_{ij} is the weight between the i'th feature and the j'th neuron. If the hidden-layer contains J neurons, then the vector of features transformed by the hidden layer, denoted $\boldsymbol{h}_n$, can be simplified to:

$$\boldsymbol{h}_n = \boldsymbol{f}\left(W\boldsymbol{x}_n + w_0 \mathbf{1}_J\right), \tag{2}$$

where $\mathbf{1}_J$ is a vector of one's of dimension J.

Assuming the activation in the output-layer is linear, then the expected response of the n'th observation is:

$$\hat{y}_n = \boldsymbol{d}_n^T \boldsymbol{\beta}, \tag{3}$$

where $\boldsymbol{d}_n$ is either the output of the hidden-layer (i.e. $\boldsymbol{d}_n = \boldsymbol{h}_n$), or the concatenation of input-layer and the output of the hidden-layer (i.e. $\boldsymbol{d}_n = [\boldsymbol{h}_n^T \ \boldsymbol{x}_n^T]^T$), dependent on whether the functional link is inactive or active, respectively. Furthermore, in the following, references to the the functional link will be avoided when possible, and the dimension of the features $\boldsymbol{d}_n$ will simply be denoted l.

The **RWNN** package allows the user to specify the number of neurons in the hidden-layer, and the activation function among those listed in Table 1.

Thus, if the weights of the hidden-layer can be assumed to be known, then the expected response of an RWNN is identical to that of a linear model. However, before estimating the weights, it it is necessary to determine the weights of the hidden-layer.

Sampling hidden-weights

In RWNN's, the standard approach to determining the weights between in input- and hidden-layer(s), is to sample the weights randomly from a uniform distribution limited to an appropriate interval for the activation function used in the hidden-layer. However, as shown by

Table 1: The name, specification, function, and range for each of the activation functions implemented in the **RWNN** package.

Name	Specification	Function	Range
Identity	`identity`	$f(x) = x$	$(-\infty, \infty)$
Bent identity	`bentidentity`	$f(x) = \dfrac{\sqrt{x^2+1}-1}{2} + x$	$(-\infty, \infty)$
Gaussian	`gaussian`	$f(x) = \exp(-x^2)$	$(0, 1]$
Hyperbolic tangent	`tanh`	$\tanh(x) = \dfrac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$	$(-1, 1)$
Rectified linear unit	`relu`	$f(x) = \max\{0, x\}$	$[0, \infty)$
Sigmoid	`sigmoid`	$f(x) = \dfrac{1}{1+\exp(-x)}$	$(0, 1)$
Sigmoid linear unit	`silu`	$f(x) = \dfrac{x}{1+\exp(-x)}$	$[-0.278..., \infty)$
Softsign	`softsign`	$f(x) = \dfrac{x}{1+\lvert x\rvert}$	$(-1, 1)$
Softplus	`softplus`	$f(x) = \ln(1+x)$	$(0, \infty)$
SQNL	`sqnl`	$f(x) = \begin{cases} -1, & \text{if } x < -2 \\ x + \frac{x^2}{4}, & \text{if } -2 \leq x < 0 \\ x - \frac{x^2}{4}, & \text{if } 0 \leq x < 2 \\ 1, & \text{if } 2 \leq x \end{cases}$	$[-1, 1]$
Squared RBF	`sqrbf`	$f(x) = \begin{cases} 1 - \frac{x^2}{2}, & \text{if } \lvert x\rvert \leq 1 \\ \frac{(2-\lvert x\rvert)^2}{2}, & \text{if } 1 < \lvert x\rvert < 2 \\ 0, & \text{if } \lvert x\rvert \geq 2 \end{cases}$	$[0, 1]$

(Wang and Liu 2017), using complete random initialisation might not be the best approach when generating these weights, as it tends to require a large number of neurons in the hidden-layer for the RWNN to have generalisability. Therefore, they introduced two alternatives to the random initialisation: quasi-random numbers, and random orthogonal projections. The three methods have all been implemented in the **RWNN** package, and are outlined below:

- **Random initialisation**: The weights between the input- and hidden-layer, w_{ij}, are sampled independently from the same distribution, $\mathcal{D}(\theta)$. The **RWNN** package is set-up to facilitate the use of any (including user defined) distribution to sample these weights. However, if nothing is specified, it defaults to a uniform distribution on the interval $[-1; 1]$, i.e. $w_{ij} \sim \mathcal{U}([-1; 1])$ for all i and j.
- **Quasi-random initialisation**: Quasi-random numbers have the distinct advantage, when compared to *true* random numbers, that they will cover the domain of interest faster and more evenly. Thus, when using quasi-random numbers, the space can be more effectively explored, leading to a higher performance using less neurons in the hidden-layer. The **RWNN** package allows for the use of Halton and Sobol' sequences to

generate quasi-random numbers by using the R package: **randtoolbox** (Christophe and Petr 2023). If nothing is specified, it defaults to creating quasi-random numbers on the cube $[-1; 1] \times [-1; 1]$.

- **Random orthogonal initialisation** (default): The idea is similar to that of random initialisation, but instead of using the randomly initialized weights directly, it uses an orthonormal basis of the sampled weights. The method used in the **RWNN** package corresponds to Algorithm 1 of (Wang and Liu 2017).

Estimating output-weights

Given a matrix of features, D, (i.e. a matrix where the n'th row contains $\boldsymbol{d}_n^T$) and a vector containing the targets, $\boldsymbol{y}$, the output-weights $\boldsymbol{\beta}$ can be found by minimising sum-of-squared errors (SSE):

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \Big\{ ||\boldsymbol{y} - D\boldsymbol{\beta}||_2^2 \Big\}. \tag{4}$$

In cases where $N > l$, the solution to this system of equations can be found by the normal equation, as:

$$\hat{\boldsymbol{\beta}}^{(ols)} = (D^T D)^{-1} D^T \boldsymbol{y}. \tag{5}$$

However, when the number of columns of D is bigger than the number of observations (i.e. $l >> N$), this optimisation problem becomes ill-posed. In such cases, the two most common approaches, to estimating the output-weights, are Moore-Penrose pseudoinverse and regularisation.

Using Moore-Penrose pseudoinverse (Bjerhammar 1951, Penrose (1955)), D^+, the solution to the optimisation problem is given by:

$$\hat{\boldsymbol{\beta}}^{(pseudo)} = D^+ \boldsymbol{y}. \tag{6}$$

Regularisation of the SSE, seen in Eq. (4), is achieved by penalising the size of the output-weights. This penalisation is usually performed using either the ℓ_1 or ℓ_2-norm, creating the following optimisation problem:

$$\hat{\boldsymbol{\beta}}^{(reg)} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \Big\{ ||\boldsymbol{y} - D\boldsymbol{\beta}||_2^2 + \lambda ||\boldsymbol{\beta}||_q^q \Big\}, \tag{7}$$

where $q \in \{1, 2\}$, and λ is a penalisation constant. The penalisation constant should be chosen such that it minimises the out-of-sample error (using e.g. k-fold cross-validation during the training process).

Estimating of the weights differs slightly dependent on the choice of q:

- $\underline{\boldsymbol{q = 1}}$:
  The optimisation problem in Eq. (7) is equivalent to performing lasso-regression (Santosa and Symes 1986), (Tibshirani 1996). Unlike ridge-regression ($q = 2$) it is not possible to find a closed form solution to the optimisation problem and, thereby, the lasso estimated output-weights, $\hat{\boldsymbol{\beta}}^{(lasso)}$. However, it has been shown by (Friedman, Hastie, Höfling, and Tibshirani 2007), that $\hat{\boldsymbol{\beta}}^{(lasso)}$ can be found using coordinate descent.

- $\underline{\boldsymbol{q = 2}}$:
  The optimisation problem in Eq. (7) is equivalent to performing ridge-regression on the concatenated features, instead of the input features, and the solution can be found as (Hoerl and Kennard 2000):

$$\hat{\boldsymbol{\beta}}^{(ridge)} = \left(D^T D + \lambda I_l\right)^{-1} D^T \boldsymbol{y}, \tag{8}$$

  where I_l is the identity matrix of size l.

2.2. Deep RWNN

Extending FFNN's from a single hidden-layer to K hidden-layers will barely change the governing equation of each neuron; assuming $h_{ni}^{(0)}$ is the i'th feature x_{ni} and that the number of neurons in the k'th layer is $J^{(k)}$, then the output of the j'th neuron in the k'th layer for the n'th observation, is given by:

$$h_{nj}^{(k)} = f^{(k)} \left(\sum_{i=1}^{J^{(k-1)}} w_{ij}^{(k)} h_{ni}^{(k-1)} + w_0^{(k)} \right), \tag{9}$$

where $f^{(k)}$ is the activation function, $w_0^{(k)}$ is the bias, and $w_{ij}^{(k)}$ is the weight between the i'th neuron and the j'th neuron in the k'th layer for $k = 1, ..., K$. Using vector notation this is simplified to:

$$\boldsymbol{h}_n^{(k)} = \boldsymbol{f}^{(k)} \left(W^{(k)} \boldsymbol{h}^{(k)} + w_0^{(k)} \mathbf{1}_k \right), \tag{10}$$

where $\mathbf{1}_k$ is a vector of one's of dimension J^k.

Using deep FFNN's as starting point, when creating deep RWNN's two approaches can be taken to predicting the target: (1) use the output of just the last hidden-layer, or (2) concatenate the outputs of all K hidden-layers. The graphical interpretation of the difference between the two approaches can be seen on the left and right-hand sides of Figure 2, respectively.

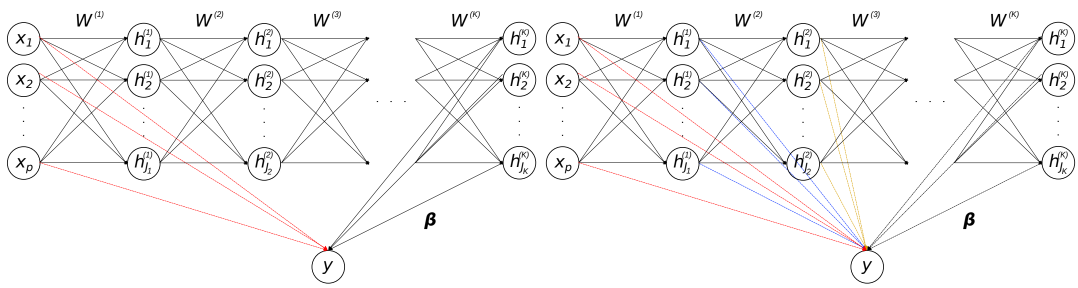

Figure 2: Graph representation of a deep random weight neural network (RWNN) with functional link between the input-layer, intermediate hidden-layers, and the output-layer.

The two structures depicted in Figure 2, yields four options when constructing a deep RWNN:

- $\boldsymbol{d}_n = \boldsymbol{h}_n^{(K)}$
- $\boldsymbol{d}_n = [(\boldsymbol{h}_n^{(K)})^T \ \boldsymbol{x}_n^T]^T$

- $\boldsymbol{d}_n = [(\boldsymbol{h}_n^{(1)})^T \ (\boldsymbol{h}_n^{(2)})^T \ \cdots \ (\boldsymbol{h}_n^{(K)})^T]^T$
- $\boldsymbol{d}_n = [(\boldsymbol{h}_n^{(1)})^T \ (\boldsymbol{h}_n^{(2)})^T \ \cdots \ (\boldsymbol{h}_n^{(K)})^T \ \boldsymbol{x}_n^T]^T$

Given a matrix D, with its n'th row corresponding to $\boldsymbol{d}_n$, it follows that estimating the output-weights of a deep RWNN is equivalent to that of the RWNN using only a single hidden-layer.

Like the estimation process, sampling the hidden weights is similar to that of RWNN's with a single hidden-layer. However, while the random and random orthogonal initialisation methods can be performed independently on a layer-by-layer basis, the quasi-random initialisation has to continue its quasi-random sequence layer-to-layer (i.e. the sequence is not reset between layers).

The **RWNN** package allows the user to specify the number of neurons and the activation function (found in Table 1) in each of the K hidden-layers, as well as which of the four modelling approaches should be taken (when nothing is specified it defaults to the third option).

2.3. Sparse RWNN

Pre-trained using an auto-encoder

Pruning using magnitude

Pruning using Fisher information

Pruning using mutual information

2.4. Ensemble methods

Stacking

Bagging

Boosting

Ensemble Deep RWNN

3. Examples

4. Conclusions

References

Bjerhammar A (1951). *Application of calculus of matrices to method of least squares : with special reference to geodetic calculations.* Acta polytechnica. Elander.

Breiman L (1996a). "Bagging Predictors." *Machine Learning*, **24**, 123–140.

Breiman L (1996b). "Stacked regressions." *Machine Learning*, **24**, 49–64.

Breiman L (2001). "Random Forests." *Machine Learning*, **45**, 5–32.

Christophe D, Petr S (2023). *randtoolbox: Generating and Testing Random Numbers*. R package version 2.0.4.

Eddelbuettel D, François R (2011). "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08.

Eddelbuettel D, Sanderson C (2014). "RcppArmadillo: Accelerating R with high-performance C++ linear algebra." *Computational Statistics and Data Analysis*, **71**, 1054–1063. URL http://dx.doi.org/10.1016/j.csda.2013.02.005.

Friedman J, Hastie T, Höfling H, Tibshirani R (2007). "Pathwise coordinate optimization." *The Annals of Applied Statistics*, **1**(2), 302 – 332.

Friedman JH (2001). "Greedy function approximation: A gradrient boosting machine." *The Annals of Statistics*, **29**, 1189–1232.

Gosso A (2012). *elmNN: Implementation of ELM (Extreme Learning Machine) algorithm for SLFN (Single Hidden Layer Feedforward Neural Networks).*

Henríquez PA, Ruz GA (2018). "Twitter Sentiment Classification Based on Deep Random Vector Functional Link." In *2018 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–6.

Hoerl AE, Kennard RW (2000). "Ridge Regression: Biased Estimation for Nonorthogonal Problems." *Technometrics*, **42**(1), 80–86.

Huang GB, Zhu QY, Siew CK (2006). "Extreme learning machine: Theory and applications." *Neurocomputing*, **70**(1), 489–501.

Kourentzes N (2019). *nnfor: Time Series Forecasting with Neural Networks*. R package version 0.9.6, URL https://CRAN.R-project.org/package=nnfor.

Meyer D, Dimitriadou E, Hornik K, Weingessel A, Leisch F (2023). *e1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien*. R package version 1.7-14, URL https://CRAN.R-project.org/package=e1071.

Mouselimis L (2022). *elmNNRcpp: The Extreme Learning Machine Algorithm*. R package version 1.0.4, URL https://CRAN.R-project.org/package=elmNNRcpp.

Pao Y, Park G, Sobajic D (1992). "Learning and generalization characteristics of random vector Functional-link net." *Neurocomputing*, **6**, 163–180.

Penrose R (1955). "A generalized inverse for matrices." *Mathematical Proceedings of the Cambridge Philosophical Society*, **51**(3), 406–413. doi:10.1017/S0305004100030401.

R Core Team (2021). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Santosa F, Symes WW (1986). "Linear Inversion of Band-Limited Reflection Seismograms." *SIAM Journal on Scientific and Statistical Computing*, **7**(4), 1307–1330.

Schmidt W, Kraaijveld M, Duin R (1992). "Feedforward neural networks with random weights." In *Proceedings., 11th IAPR International Conference on Pattern Recognition. Vol.II. Conference B: Pattern Recognition Methodology and Systems*, pp. 1–4.

Shi Q, Katuwal R, Suganthan P, Tanveer M (2021). "Random vector functional link neural network based ensemble deep learning." *Pattern Recognition*, **117**, 107978.

Sui X, He S, Vilsen SB, Teodorescu R, Stroe DI (2021). "Fast and Robust Estimation of Lithium-ion Batteries State of Health Using Ensemble Learning." In *2021 IEEE Energy Conversion Congress and Exposition (ECCE)*, pp. 1–8. (accepted).

Tibshirani R (1996). "Regression Shrinkage and Selection via the Lasso." *Journal of the Royal Statistical Society. Series B (Methodological)*, **58**(1), 267–288.

Wang W, Liu X (2017). "The selection of input weights of extreme learning machine: A sample structure preserving point of view." *Neurocomputing*, **261**, 28 – 36.

Wolpert D (1992). "Stacked generalization." *Neural Networks*, **5**, 241–259.

Zhang Y, Wu J, Cai Z, Du B, Yu PS (2019). "An unsupervised parameter learning model for RVFL neural network." *Neural Networks*, **112**, 85–97.

Affiliation:

Søren B. Vilsen
Department of Mathematical Sciences, Aalborg University
Skjernvej 4A,
9220 Aalborg East
E-mail: svilsen@math.aau.dk
URL: people.math.aau.dk/~svilsen

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd