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.

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 regression 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. The RWNN is implemented in C++ through **Rcpp** and **RcppArmadillo**, and 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.
- **deep RWNN** (Henríquez and Ruz 2018): 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 extension of deep RWNNs using the output of each hidden layer to create separate RWNNs. These RWNNs are then used to create an ensemble prediction of the target.

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

- **Stacking**: Stack multiple randomly generated RWNN's, and estimate their contribution to the weighted ensemble prediction using k-fold cross-validation.
- **Bagging** (Sui, He, Vilsen, Teodorescu, and Stroe 2021): Bootstrap aggregation of 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 can be chosen at random.
- **Boosting**: Gradient boosting creates a series of RWNN's, where an element, k, of the series is trained on the residual of the previous $(k-1)$ RWNN's. It further allows for manipulation of the learning rate used to improve the generalisation of the boosted model. Lastly, like the implemented bagging method, the number of features used in each iteration can be chosen at random (also called stochastic gradient boosting).

Two Bayesian sampling approaches have also been implemented using either a Metropolis-Hastings or Approximate Bayesian Computation, to sample both hidden weights and output weights of the RWNN. This sampling process is a posthoc procedure, i.e., it is applied after the initial assignment of random weights in the hidden-layers, and estimation of the output-weights.

Lastly, the **RWNN** package also includes a simple method for grid based hyperparameter optimisation, where k-fold cross-validation is applied to training-set to find the hyperparameters yielding the smallest cross-validation error on the supplied grid.

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

The general structure of an RWNN with a single hidden-layer can be seen in Figure 1. The RWNN is a simplification of a simple FFNN, where the weights between the input-layer and the hidden-layer are kept fixed after the randomly initialisation of the network. That is, the weights between the last hidden-layer and the output-layer are the only weights estimated during the training process. Furthermore, an RWNN may include a direct (also called functional) link between the features and the output, shown as dashed lines in Figure 1. When these links are active, the output can be seen as a concatenation of the last hidden-layer and the input-layer (i.e. a concatenation of the both the original and random non-linear transformation of the features). Using this simplification, the estimation of the output-weights simplifies greatly if the activation between the concatenated layer and the output is linear (i.e. the identity function), as estimating of the output-weights becomes equivalent to estimating the weights in a (regularised) multiple linear regression.

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 output of observation n, then the output of the j'th neuron of the

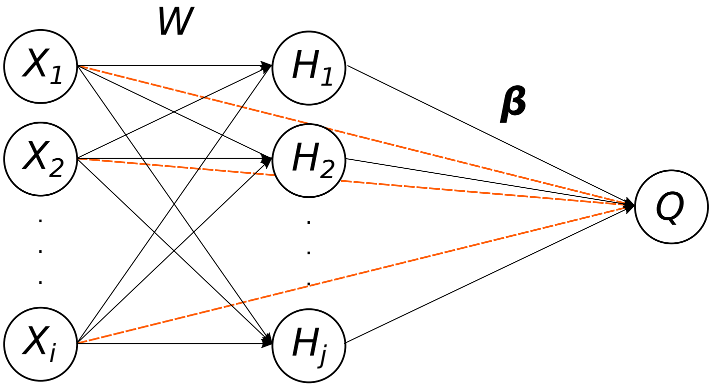

Figure 1: Graph representation of a random weight neural network (RWNN) with funcitonal link between the input and the output layer.

hidden layer:

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

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 in n'th observation. If the hidden-layer contains J neurons the vector, the vector of transformed features of the hidden layer, denoted $\boldsymbol{h}_n$, simplifying the notation of Eq. (8) to:

$$\boldsymbol{h} = \boldsymbol{f}\left(WX + w_0\mathbf{1}\right), \tag{2}$$

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

Let $\boldsymbol{d}_n = [\boldsymbol{h}_n^{(K)}\ \boldsymbol{x}_n]^T$ be a stacked vector of the features transformed by the K hidden-layers, and original features. Furthermore, assuming the activation in the output-layer is linear, then the predicted response of the n'th observation is:

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

Let D be the matrix of the concatenated features (i.e. the n'th row contains $\boldsymbol{d}_n$) and $\boldsymbol{y}$ be a vector containing the output, then the output-weights $\boldsymbol{\beta}$ can be found by minimising sum-of-squared errors (SSE):

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

However, in many application the number of concatenated features (i.e. the number of columns of D) is much bigger than the number of observations, i.e. $(p + J^{(K)}) > N$. The two most common approaches to estimating the parameters in cases where $(p + J^{(K)}) > N$ are the Moore-Penrose pseudoinverse and regularisation.

When using Moore-Penrose pseudoinverse (Bjerhammar 1951, Penrose (1955)), D^+, the solution to the optimisation problem is simply:

$$\hat{\boldsymbol{\beta}}^{(mp)} = D^+\boldsymbol{y}. \tag{5}$$

When regularisation in introduced the SSE, seen in Eq. (4), is penalised by the size of the output-weights. The penalisation is either performed using ℓ_1 or ℓ_2 norm, creating the following optimisation problem:

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \Big\{ ||\boldsymbol{y} - D\boldsymbol{\beta}||_2^2 + \lambda ||\boldsymbol{\beta}||_q^q \Big\}, \tag{6}$$

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 training process).

If $q = 1$, the optimisation problem in Eq. (6) is equivalent to lasso regression (Santosa and Symes 1986, Tibshirani (1996)). Unlike ridge-regression it is not possible to find a closed form solution of the lasso estimated output-weights, $\hat{\boldsymbol{\beta}}^{(lasso)}$, however, they can be found by using coordinate descent (Friedman, Hastie, Höfling, and Tibshirani 2007).

If $q = 2$, the optimisation problem in Eq. (6) is equivalent to ridge-regression using the concatenated features, instead of the input features, and the solution can be found as (Hoerl and Kennard 2000):

$$\hat{\boldsymbol{\beta}}^{(ridge)} = \left(D^T D + \lambda I_{p+J^{(K)}} \right)^{-1} D^T \boldsymbol{y}, \tag{7}$$

where $I_{p+J^{(K)}}$ is the identity matrix of size $p + J^{(K)}$.

2.1. Deep RWNN

$$h_{nj}^{(k)} = f^{(k)} \left(\sum_{i=1}^{p} w_{ij}^{(k)} h_{ni}^{(k-1)} + w_0^{(k)} \right), \tag{8}$$

for $k = 1, ..., K$, where $h_{ni}^{(0)}$ is feature x_{ni}, $f^{(k)}$ is the activation function, $w_0^{(k)}$ is the bias, and $w_{ij}^{(k)}$ is the weight between the i'th feature and the j'th neuron in the k'th layer. If the hidden-layer contains $J^{(k)}$ neurons the vector, the vector of transformed features of the k'th layer is denoted $\boldsymbol{h}_n^{(k)}$ simplifying the notation of Eq. (8) to:

$$\boldsymbol{h}^{(k)} = \boldsymbol{f}^{(k)} \left(W^{(k)} \boldsymbol{h}^{(k)} + w_0^{(k)} \mathbf{1} \right), \tag{9}$$

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

Let $\boldsymbol{d}_n = [\boldsymbol{h}_n^{(K)} \; \boldsymbol{x}_n]^T$ be a stacked vector of the features transformed by the K hidden-layers, and original features. Furthermore, assuming the activation in the output-layer is linear, then the predicted response of the n'th observation is:

$$\hat{y}_n = \boldsymbol{d}_n^T \boldsymbol{\beta} \tag{10}$$

2.2. Sparse RWNN

2.3. Ensemble methods

Stacking

Bagging

Boosting

2.4. Ensemble Deep RWNN

2.5. Bayesian methods

Metropolis-Hastings sampling

Approximate Bayesian Computation

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.

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.

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.

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.

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