Using the 'neuralnet' Package [R]

Spyros Orfanos

Concordia University, Summer 2019

1. Introduction

The neuralnet package is an R package used to train feed-forward neural networks. The purpose of this paper is to provide further background information, insight, and documentation to the workings and usage of the 'neuralnet' package. An example is provided to demonstrate how to use various functions from this package. The results from this example are then validated with a feed-forward neural network that was trained using first principles.

2. Training Algorithms

When training a neural network, the goal is to optimize the weights such that the objective function (cost-function) is minimized. This is typically done via gradient descent on the the objective function. Gradient descent involves moving towards a local minimum of a function by taking steps in the direction opposite to the function's gradient. More formally, for some vector x_0 and function F(x), if $x_1 = x_0 - \alpha \cdot \nabla \text{ F}(x_0)$, then, for small enough α, $\text{F}(x_1) \leq \text{F}(x_0)$.

The objective function of a neural network is highly non-linear, so computing its gradient can be a cumbersome process. With the neuralnet function, the user may train a neural network using backpropagation, resilient backpropagation (with or without weight backtracking), or globally convergent algorithms (sag or sal). These algorithms provide efficient ways to compute the gradient and/or alternatives to gradient descent.

2.1. Backpropagation

From the chain rule of calculus, we have:

$$\frac{\partial Error}{\partial w_{ij}^k} = \frac{\partial Error}{\partial a_1^L} \cdot \frac{\partial a_1^L}{\partial w_{ij}^k} = \frac{\partial Error}{\partial a_1^L} \cdot \frac{\partial a_1^L}{\partial a_j^l} \cdot \frac{\partial a_j^l}{\partial w_{ij}^k} \tag{1}$$

The Backpropagation Algorithm offers a convenient way to calculate $\frac{\partial a_1^L}{\partial a_k^s}$ recursively using $\frac{\partial a_1^L}{\partial a_k^{s+1}}$ which is required to compute the gradient of the error function [1]. The algorithm has its name as it propagates backwards through the neural network to sequentially compute the necessary gradients [2]. By doing this, we can calculate the gradient of the error function, and then apply gradient descent to update the weights and improve the model's performance.

2.2. Resilient Backpropagation

The logistic sigmoid and hyperbolic tangent activation functions have led to much success in deep learning; however, they are susceptible saturation, which results in extremely slow convergence. These functions are said to saturate because as the magnitude of the input becomes very large (or small), the partial derivatives at these points approach zero. Thus, using gradient descent to train the model would converge extremely slowly. The Resilient Backpropagation (RPROP) algorithm offers a solution to this pitfall: only the signs of the partial derivatives are used to determine the directions in which the weights will change. The size of the change for each weight is determined by a separate factor, $\Delta_{i,j}$, which is calculated at each iteration as follows:

$$\Delta_{ij}^{(t)} = \begin{cases} \max(n^+ \cdot \Delta_{ij}^{(t-1)}, \Delta_{max}) & \text{if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ \min(n^- \cdot \Delta_{ij}^{(t-1)}, \Delta_{min}) & \text{if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \\ \Delta_{ij}^{(t-1)} & \text{otherwise} \end{cases} \tag{2}$$

$$\text{for } 0 < n^- < 1 < n^+$$

We see that if two if two consecutive partial derivative are of the same sign, Δ_{ij} is increased in order to accelerate convergence to a local minimum. Conversely, if two consecutive partial derivatives are of the opposite sign, the RPROP algorithm assumes the local minimum has been crossed, and hence reduces the step size to avoid crossing the local minimum again. After each calculation of Δ_{ij}, the weights are updating according to the weight-update rule. There are two variants to the weight-update rule of the RPROP algorithms: with weight backtracking ($RPROP^+$) and without weight backtracking ($RPROP^-$).

2.2.1. $RPROP^+$

The update rule for w_{ij} in the RPROP with weight backtracking algorithm is given by:

$$w_{ij}^{(t+1)} = \begin{cases} w_{ij}^{(t)} \text{ - } \Delta_{ij}^{(t)} \cdot sign(\frac{\partial E}{\partial w_{ij}}^{(t)}) & \text{if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t)} > 0 \\ w_{ij}^{(t-1)} \text{ and set } \frac{\partial E}{\partial w_{ij}}^{(t)} = 0 & \text{if } \frac{\partial E}{\partial w_{ij}}^{(t-1)} \cdot \frac{\partial E}{\partial w_{ij}}^{(t)} < 0 \end{cases} \tag{3}$$

In case of a change of sign of the partial derivative, the previous weight update is reverted. Setting the stored derivative to zero avoids an update of the learning rate in the next iteration.

2.2.2. $RPROP^-$

The update rule for w_{ij} in the RRPOP without weight backtracking algorithm is given by:

$$w_{ij}^{(t+1)} = w_{ij}^{(t)} \text{ - } \Delta_{ij}^{(t)} \cdot sign(\frac{\partial E}{\partial w_{ij}}^{(t)}) \tag{4}$$

If two consecutive partial derivatives both have the same sign, a larger step is taken in the direction opposite to the gradient since the direction of the gradient is still the same

(i.e., we under-stepped, so keep moving in that direction). Conversely, if two consecutive partial derivatives are of opposite signs, then the step size is decreased since the direction of the gradient has changed (i.e., we overstepped, so take a small step in the new direction).

2.3. Globally Convergent Algorithms

Two globally convergent variations of the $RPROP^-$ algorithm are presented: sag and slr. These variations are designed to converge to a local minimizer of a nonlinear function, from almost any starting point. modifies one learning rate, either the learning rate associated with the smallest absolute gradient (sag) or the smallest learning rate (slr) itself. The learning rates in the grprop algorithm are limited to the boundaries defined in learningrate.limit".

3. 'neuralnet' Function

The neuralnet function can be used to train a multi-layered neural network, and then returns an object of type 'nn'. Here we explain how to train a neural network, understand the results, and make predictions on new data using this function.

3.1. Arguments

Here is a list of arguments of the neuralnet function (only the first three are required):

- *formula*: a symbolic description of the model to be fitted. Note: response variable can have more than one column (eg: y = cbind(y1, y2)).
- *data*: a data frame containing the variables specified in formula.
- *hidden*: a vector of integers specifying the number of hidden neurons in each layer.
- *algorithm*: a string containing the algorithm type to calculate the neural network (eg: 'bprop', 'rprop+', etc.).
- *threshold*: a numeric value specifying the threshold for the partial derivatives of the error function as stopping criteria.
- *stepmax*: the maximum steps for the training of the neural network. Reaching this maximum leads to a stop of the neural network's training process.
- *rep*: the number of repetitions for the neural network's training.
- *startweights*: a list containing starting values for the weights and biases. It is of the same form as $weights.
- *learningrate*: a numeric value specifying the learning rate used by traditional backpropagation. Used only for traditional backpropagation.
- *learningrate.limit*: a vector or a list containing the lowest and highest limit for the learning rate. Used only for RPROP and GRPROP.

- *learningrate.factor*: a vector or a list containing the multiplication factors for the upper and lower learning rate. Used only for RPROP and GRPROP.

- *lifesign*: a string specifying how much the function will print during the calculation of the neural network. 'none', 'minimal' or 'full'.

- *lifesign.step*: an integer specifying the stepsize to print the minimal threshold in full lifesign mode.

- *err.fct*: a differentiable function that is used for the calculation of the error. Alternatively, the strings 'sse' (Sum of S and 'ce' which stand for the sum of squared errors and the cross-entropy can be used.

- *act.fct*: a differentiable function that is used for smoothing the result of the cross product of the covariate or neurons and the weights. Additionally the strings, 'logistic' and 'tanh' are possible for the logistic function and tangent hyperbolicus.

- *linear.output*: T/F switch if act.fct should not be applied to the output neurons.

- *exclude*: a vector or a matrix specifying the weights, that are excluded from the calculation. If given as a vector, the exact positions of the weights must be known. A matrix with n-rows and 3 columns will exclude n weights, where the first column stands for the layer, the second column for the input neuron and the third column for the output neuron of the weight.

- *constant.weights*: a vector specifying the values of the weights that are excluded from the training process and treated as fix.

- *likelihood*: T/F switch. If the error function is equal to the negative log-likelihood function, the information criteria AIC and BIC will be calculated. Furthermore the usage of confidence.interval is meaningful.

3.2. Results

Note that many important parameters and results of the trained neural network are accessible directly and via 'result.matrix'.

- *$weights* the final weights of the neural network.

- *$result.matrix$error* the training error. Note: SSE is defined as $\sum_{i=1}^{n} \frac{1}{2}(y_i - \hat{y}_i)^2$

- *$result.matrix$reached.threshold* : returns the largest partial derivative of the cost function on the final iteration. Recall that this is a stopping criteria, so if the threshold is below the desired threshold, training ends.

- *$result.matrix$steps*: the number of steps until the threshold is reached.

- $result.matrix$intercept.to.ilayerhidj : bias of the j^{th} neuron in the i^{th} layer, i.e., $w^i_{1,j}$. When plotting the neural network, these weights are represented by the blue arrows.
- $result.matrix$x.to.klayhidj : Accesses weight of the j^{th} neuron in the input layer, i.e., $w^k_{i,j}$. When plotting the neural network, these weights are represented by the black arrows.

3.3. 'predict' Function

The 'predict' function is used to make predictions on new data using the trained neural network. It is as simple as typing the command 'predict(myNN, newdata = someData)'.

3.4. 'plot' Function

The 'plot' function can take an object of type 'nn' as an argument, so this function can be used to visualize the structure of a given neural network. Weights are represented by black lines, and biases are represented by blue lines. By default, the values of the weights and biases are displayed, as well as the training error and the number of steps until convergence.

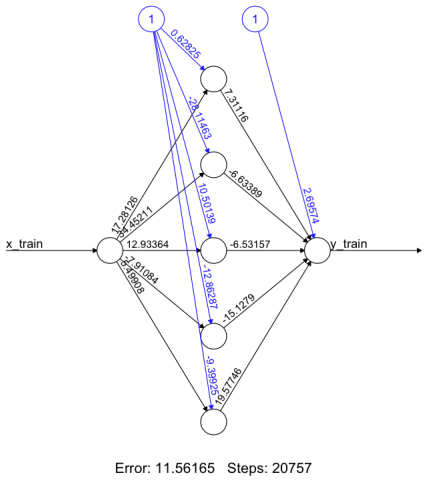

Figure 1: Visualization of a 1-5-1 ANN

4. An Example to Validate Results

Here, we validate some of the results of the neuralnet package by comparing it with another neural feed-forward network that was trained without the use of packages. Both neural networks have the same training data, hyper-parameters (hidden units: 8, layers: 2), cost function (SSE, no decay), learning algorithm (back-propagation with LR = 0.01) and stopping criteria (threshold = 0.01). Training and testing data was generated from the function y = $x^3 - 4 \cdot sin(4 \cdot x) + Z$, where Z is white noise with variance 1.

The first neural network, trained using the neuralnet package, satisfied the stopping criteria after approximately 20,000 steps with a threshold of 9.11e-03 and training SSE of 10.22. After 20,000 steps, the second neural network, trained without the neuralnet package,

had a threshold of 1.79e-03 and training SSE of 11.86. On new data, the first neural network had test SSE of 9.52 and the second one had test SSE of 7.82. Thus, all the results are similar and this package works as expected. The results are summarized in the following figures:

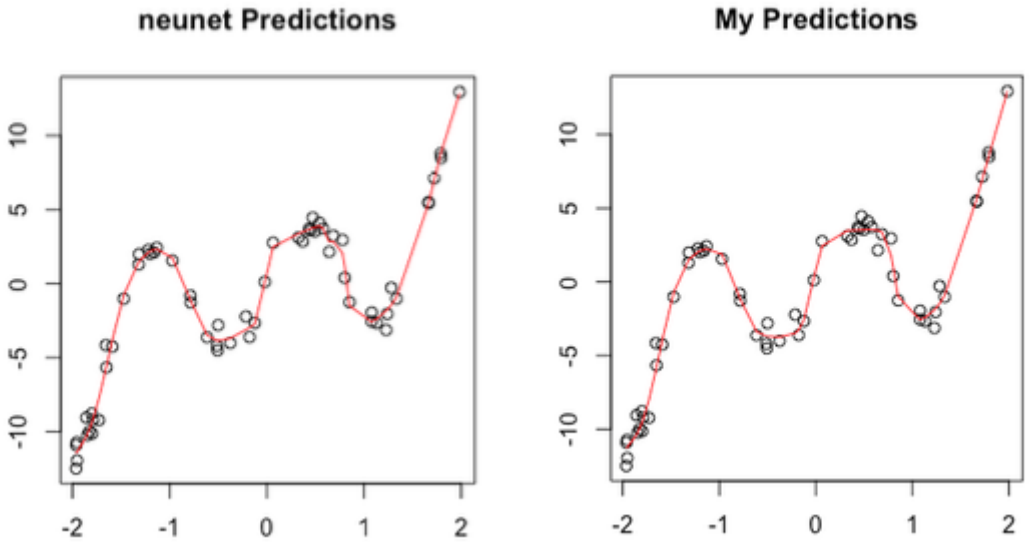

Figure 2: Predictions with neural networks

5. References

[1] Zhang Z. Neural networks: further insights into error function, generalized weights and others. Ann Transl Med. 2016;4(16):300. doi:10.21037/atm.2016.05.37

[2] F. Godin. STAT497: Reinforcement Learning, Supplementary Material on Neural Networks, Concordia University, 2019.

[3] C. Igel and M. Huesken. Improving the Rprop Learning Algorithm. Proceedings of the Second International Symposium on Neural Computation, NC'2000, pp. 115–121, ICSC Academic Press, 2000

[4] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In Proceedings of the IEEE In- ternational Conference on Neural Networks, pages 586–591. IEEE Press, 1993.