Chapter 2

Bayesian Regularized Neural Networks for Small *n* Big *p* Data

Hayrettin Okut

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/63256

Abstract

Artificial neural networks (ANN) mimic the function of the human brain and they have the capability to implement massively parallel computations for mapping, function approximation, classification, and pattern recognition processing. ANN can capture the highly nonlinear associations between inputs (predictors) and target (responses) variables and can adaptively learn the complex functional forms. Like other parametric and nonparametric methods, such as kernel regression and smoothing splines, ANNs can introduce overfitting (in particular with highly-dimensional data, such as genome wide association -GWAS-, microarray data etc.) and resulting predictions can be outside the range of the training data. Regularization (shrinkage) in ANN allows bias of parameter estimates towards what are considered to be probable. Most common techniques of regularizations techniques in ANN are the Bayesian regularization (BR) and the early stopping methods. Early stopping is effectively limiting the used weights in the network and thus imposes regularization, effectively lowering the Vapnik-Chervonenkis dimension. In Bayesian regularized ANN (BRANN), the regularization techniques involve imposing certain prior distributions on the model parameters and penalizes large weights in anticipation of achieving smoother mapping.

Keywords: artificial neural network, Bayesian regularization, shrinkage, *p*>>*n*, prediction ability

1. Introduction

The issue of dimensionality of independent variables (i.e. when the number of observations is comparable to or larger than the sample size; small *n* big *p; p*>>*n*) has garnered much attention

over the last few years, primarily due to the fact that high-dimensional data are so common in up-to-date applications (e.g. microarray data, fMRI image processing, next generation sequencing, and many others assays of social and educational data). This issue is of particular interest in the field of human molecular genetics as the growing number of common single nucleotide polymorphisms (minor allele frequency > 0.01) available for assay in a single experiment, now close to 10 million (http://www.genome.gov/11511175), is quickly outpacing researchers' ability to increase sample sizes which typically number in the thousands to tens of thousands. Further, human studies of associations between molecular markers and any trait of interest include the possible presence of cryptic relationships between individuals that may not be tractable for use in traditional statistical models. These associations have been investigated primarily using a naïve single regression model for each molecular marker and linear regression models using Bayesian framework and some machine learning techniques typically ignoring interactions and non-linearity [1]. Soft computing techniques have also been used extensively to extract the necessary information from these types of data structures. As a universal approximator, Artificial Neural Networks (ANNs) are a powerful technique for extracting information from large data, in particular for $p>>n$ studies, and provide a computational approach with the ability to optimize the learning algorithm and make discoveries about functional forms in an adaptive approach [2, 3]. Moreover, ANNs offer several advantages, including requiring less formal statistical training, an ability to perfectly identify complex nonlinear relationships between dependent and independent variables, an ability to detect all possible interactions between input variables, and the availability of multiple training algorithms [4]. In general, the ANN architecture, or "model" in statistical jargon, is classified by the fashion in which neurons in a neural network are connected. Generally speaking, there are two different classes of ANN architecture (although each one has several subclasses). These are *feedforward* ANNs and *recurrent* ANNs. Only Bayesian regularized multilayer perceptron (MLP) *feedforward* ANNs will be discussed in this chapter.

This chapter begins with introducing of multilayer feedforward architectures. The regularization using other backpropagation algorithms to avoid overfitting will be explained briefly. Then Bayesian regularization (BR) for overfitting, Levenberg-Marquardt (LM) a training algorithm, BR, optimization of hyper parameters, inferring model parameters for given value of hyper parameters, pre-processing of data will be considered. Chapter will be ended with a MATLAB example for Bayesian Regularized feedforward multilayer artificial neural network (BRANN).

2. Multilayer perceptron feedforward neural networks

The MLP feedforward neural network considered herein is the most popular and most widely used ANN paradigm in many practical applications. The network is fully connected and divided into layers as depicted in **Figure 1**. In the left-most layer, there are input variables. The input layer consists of p_i (p=4 in **Figure 1** for illustration) independent variables and covariates. Then input layer is followed by a hidden layer, which consists of S number of neurons (S=3 in **Figure 1**), and there is a bias specific to each neuron. Algebraically, the process can be repre-

http://dx.doi.org/10.5772/63256

sented as follows. Let t_i (the target or dependent variable) be a trait (quantitative or qualitative) measured in individual i and let $p_i = \{p_{ij}\}$ be a vector of inputs (independent variables) or any other covariate measured for each individual. Suppose there are S neurons in the hidden layer. The input into neuron k (k=1, 2,…, S) prior to activation, as described in greater detail later in this chapter, is the linear function $w'_k p_i$, where $w'_k = \{w_{kj}\}$ is a vector of unknown connection strengths (slope in regression model) peculiar to neuron k, including a bias (e.g. the intercept in regression model). Each neuron in the hidden layer performs a weighted summation (n_i in **Figure 1**) of the inputs prior to activation which is then passed to a nonlinear activation function $f_k\left(b_k^{(1)} + \sum_{j=1}^{P} w_{kj} p_j\right)$. Suppose the activation function chosen in the hidden layer is the hyperbolic tangent transformation $f(x_i) = \frac{e^{x_i} - e^{-x_i}}{e^{x_i} + e^{-x_i}}$, where $f(x_i)$ is the neuron emission for input variables [3]. Based on the illustration given in the **Figure 1**, the output of each neuron (shown as purple, orange, and green nodes) in the hidden layer (t_h) with hyperbolic tangent transformation are calculated as in equation (1).

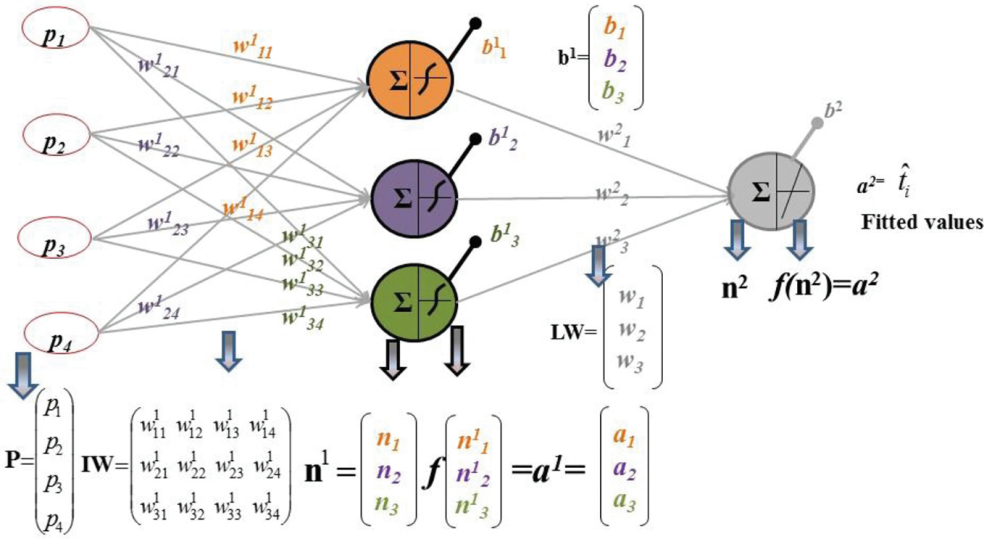


Figure 1. Artificial neural network design with 4 inputs (p_i). Each input is connected to up to 3 neurons via coefficients $w^{(l)}_{kj}$ (l denotes layer; j denotes neuron; k denotes input variable). Each hidden and output neuron has a bias parameter $b^{(l)}_j$. Here **P** = inputs, **IW** = weights from input to hidden layer (12 weights), **LW** = weights from hidden to output layer (3 weights), b^1 = Hidden layer biases (3 biases), b^2 = Output layer biases (1 bias), n^1 = **IWP** + b^1 is the weighted summation of the first layer, $a^1 = f(n^1)$ is output of hidden layer, n^2 = **LW**a^1 +b^2 is weighted summation of the second layer, and $\hat{t} = a^2 = f\left(n^2\right)$ is the predicted value of the network. The total number of parameters for this ANN is 12+3+3+1 =19.

$$t_{hidden} = \beta_0 + \beta_1\left(\frac{\exp(b_1^1 + w_{11}^1 p_1 + w_{12}^1 p_2 + w_{13}^1 p_3 + w_{14}^1 p_4) - \exp(-b_1^1 - w_{11}^1 p_1 - w_{12}^1 p_2 - w_{13}^1 p_3 - w_{14}^1 p_4)}{\exp(b_1^1 + w_{11}^1 p_1 + w_{12}^1 p_2 + w_{13}^1 p_3 + w_{14}^1 p_4) + \exp(-b_1^1 - w_{11}^1 p_1 - w_{12}^1 p_2 - w_{13}^1 p_3 - w_{14}^1 p_4)}\right) \quad \text{ORANGE}$$

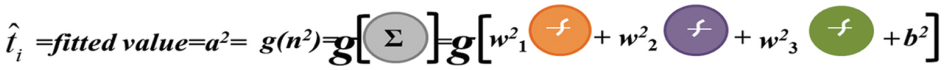
$$+\beta_2\left(\frac{\exp(b_2^1 + w_{21}^1 p_1 + w_{22}^1 p_2 + w_{23}^1 p_3 + w_{24}^1 p_4) - \exp(-b_2^1 - w_{21}^1 p_1 - w_{22}^1 p_2 - w_{23}^1 p_3 - w_{24}^1 p_4)}{\exp(b_2^1 + w_{21}^1 p_1 + w_{22}^1 p_2 + w_{23}^1 p_3 + w_{24}^1 p_4) + \exp(-b_2^1 - w_{21}^1 p_1 - w_{22}^1 p_2 - w_{23}^1 p_3 - w_{24}^1 p_4)}\right) \quad \text{PURPLE} \quad (1)$$

$$+\beta_3\left(\frac{\exp(b_3^1 + w_{31}^1 p_1 + w_{32}^1 p_2 + w_{33}^1 p_3 + w_{34}^1 p_4) - \exp(-b_3^1 - w_{31}^1 p_1 - w_{32}^1 p_2 - w_{33}^1 p_3 - w_{34}^1 p_4)}{\exp(b_3^1 + w_{31}^1 p_1 + w_{32}^1 p_2 + w_{33}^1 p_3 + w_{34}^1 p_4) + \exp(-b_3^1 - w_{31}^1 p_1 - w_{32}^1 p_2 - w_{33}^1 p_3 - w_{34}^1 p_4)}\right) \quad \text{GREEN}$$

The hidden layer outputs from neurons (orange, purple, and green) are the input of the next layer. After the activation function, the output from the hidden layer is then sent to the output layer again with weighted summation as $\sum_{k=1}^{S} w'_k f_k\left(b_k^{(1)} + \sum_{j=1}^{P} w_{kj} p_j\right) + b^{(2)}$, where w_k are weights specific to each neuron and $b^{(1)}$ and $b^{(2)}$ are bias parameters in the hidden and output layers, respectively. Finally, this quantity is again activated with the same or another activation function $g(.)$ as $g\left[\sum_{k=1}^{S} w'_k f_k(.) + b^{(2)}\right] = a^2 = \hat{t}$, which then becomes the predicted value $\hat{t}_i$ of the target variable in the training set. For instance, the predicted value $\hat{t}_i$ of the output layer based on details given in **Figure 1** and equation (1) can be calculated as,

$$\hat{t} = a^2 = g\left[\sum_{k=1}^{S} w'_k f_k(.) + b^{(2)}\right] = g\left[\begin{array}{l}\left(\frac{\exp(b_1^1 + w_{11}^1 p_1 + w_{12}^1 p_2 + w_{13}^1 p_3 + w_{14}^1 p_4) - \exp(-b_1^1 - w_{11}^1 p_1 - w_{12}^1 p_2 - w_{13}^1 p_3 - w_{14}^1 p_4)}{\exp(b_1^1 + w_{11}^1 p_1 + w_{12}^1 p_2 + w_{13}^1 p_3 + w_{14}^1 p_4) + \exp(-b_1^1 - w_{11}^1 p_1 - w_{12}^1 p_2 - w_{13}^1 p_3 - w_{14}^1 p_4)}\right) w_1^2 + \\ \left(\frac{\exp(b_2^1 + w_{21}^1 p_1 + w_{22}^1 p_2 + w_{23}^1 p_3 + w_{24}^1 p_4) - \exp(-b_2^1 - w_{21}^1 p_1 - w_{22}^1 p_2 - w_{23}^1 p_3 - w_{24}^1 p_4)}{\exp(b_2^1 + w_{21}^1 p_1 + w_{22}^1 p_2 + w_{23}^1 p_3 + w_{24}^1 p_4) + \exp(-b_2^1 - w_{21}^1 p_1 - w_{22}^1 p_2 - w_{23}^1 p_3 - w_{24}^1 p_4)}\right) w_2^2 + \\ \left(\frac{\exp(b_3^1 + w_{31}^1 p_1 + w_{32}^1 p_2 + w_{33}^1 p_3 + w_{34}^1 p_4) - \exp(-b_3^1 - w_{31}^1 p_1 - w_{32}^1 p_2 - w_{33}^1 p_3 - w_{34}^1 p_4)}{\exp(b_3^1 + w_{31}^1 p_1 + w_{32}^1 p_2 + w_{33}^1 p_3 + w_{34}^1 p_4) + \exp(-b_3^1 - w_{31}^1 p_1 - w_{32}^1 p_2 - w_{33}^1 p_3 - w_{34}^1 p_4)}\right) w_3^2 \end{array} + b^2\right] \quad (2)$$

This can be illustrated as,



The activation function applied to the output layer depends on the type of target (dependent) variable and the values from the hidden units that are combined at the output units with additional (potentially different) activation functions applied. The activation functions most widely used are the hyperbolic tangent, which ranges from 1 to 1, in the hidden layer and linear in the output layer, as is the case in our example in **Figure 1** (the sigmoidal type activation function such as tangent hyperbolic and logit in the hidden layer are used for their convenient mathematical properties and these are usually chosen as a smooth step function). If the

activation function at the output layer $g(.)$ is a linear or identity activation function, the model on the adaptive covariates $f_k(.)$ is also linear. Therefore, the regression model is entirely linear. The term "adaptive" denotes that the covariates in ANN are functions of unknown parameters (i.e. the $\{w_{kj}\}$ connection strengths), so the network can "learn" the association between independent (input) variables and target (output) variables, as is the case in standard regression models [1]. In this manner, this type of ANN architecture can also be regarded as a regression model. Of course, the level of non-linearity in ANNs is ultimately dictated by the type of activation functions used.

As depicted in **Figure 1**, a MLP feed forward architecture with one hidden layer can virtually predict any linear or non-linear model to any degree of accuracy, assuming that you have a suitable number of neurons in the hidden layer and an appropriate amount of data. But adding more neurons in hidden layers to the ANN architecture offers the model the flexibility of prediction of exceptionally complex nonlinear associations. This also holds true in function approximation, mapping, classification, and pattern recognition in approximating any nonlinear decision boundary with great precision. By adding additional neurons in the hidden layer to a MLP feedforward ANN is similar to adding additional polynomial terms to a regression model through the practice of generalization. Generalization is a method of indicating the appropriate complexity for the model in generating accurate prediction estimates based on data that is entirely separate from the training data that was used in fitting the model [5], commonly referred to as the test data set.

3. Overfitting and regularization

Like many other nonlinear estimation methods in supervised machine learning, such as kernel regression and smoothing splines, ANNs can suffer from either underfitting or overfitting. In particular, overfitting is more serious because it can easily lead to predictions that are far beyond the range of the training data [6–8]. Overfitting is unavoidable without practicing any regularization if the number of observations in the training data set is less than the number of parameters to be estimated. Before tackling the overfitting issue, let us first consider how the number of parameters in multilayer feedforward ANN is calculated. Suppose an ANN with **1000** inputs, **3** neurons in hidden and **1** neuron in output layer. The total number of parameters for this particular ANN is **3*1000** (number of weights from the input to the hidden layer) + **3** (number of biases in the hidden layer) + **3** (number of weights from the hidden to the output layer) + **1**(bias in output) = **3007**. The number of parameters for the entire network, for example, increases to **7015** when **7** neurons are assigned to the hidden layer. In these examples, either 3007 or 7015 parameters need to be estimated from 1000 observations. To wit, the number of neurons in the hidden layer controls the number of parameters (weights and biases) in the network. Determination of the optimal number of neurons to be retained in the hidden layer is an important step in the strategy of ANN. An ANN with less number of neurons may fails to capture the complex patterns between input and target variables. In contrast, an ANN with excess number of neurons in hidden layer will suffer from over-parameterization, leading to over-fitting and poor generalization ability [3]. Note that if the number of parameters in the

network is much smaller than the total number of individuals in the training data set, then that is of no concern as there is little or no chance of overfitting for given ANN. If data assigned to the training set can be increased, then there is no need to worry about the following techniques to prevent overfitting. However, in most applications of ANN, data is typically partitioned to derive a finite training set. ANN algorithms train the model (architecture) based on this training data, and the performance and generalizability of the model is gauged on how well it predicts the observations in the training data set.

Overfitting occurs when a model fits the data in the training set well, while incurring larger generalization error. As depicted in **Figure 2**, the upper panel (**Figure 2a** and **b**) shows a model which has been fitted using too many free parameters. It seems that it does an excellent fitting of the data points, as the difference between outcome and predicted values (error) at the data points is almost zero. In reality, the data being studied often has some degree of error or random noise within it. Therefore, this does not mean our model (architecture) has good generalizability for given new values of the target variable *t*. It is said that the model has a bias-variance trade-off problem. In other words, the proposed model does not reproduce the structure which we expect to be present in any data set generated by function *f(.)*. **Figure 2c** shows that, while the network seems to get better and better (i.e., the error on the training set decreases; represented by the blue line in **Figure 2a**), at some point during training (epoch 2 in this example) it truly begins to get worse again (i.e. the error on test data set increases; represented by the

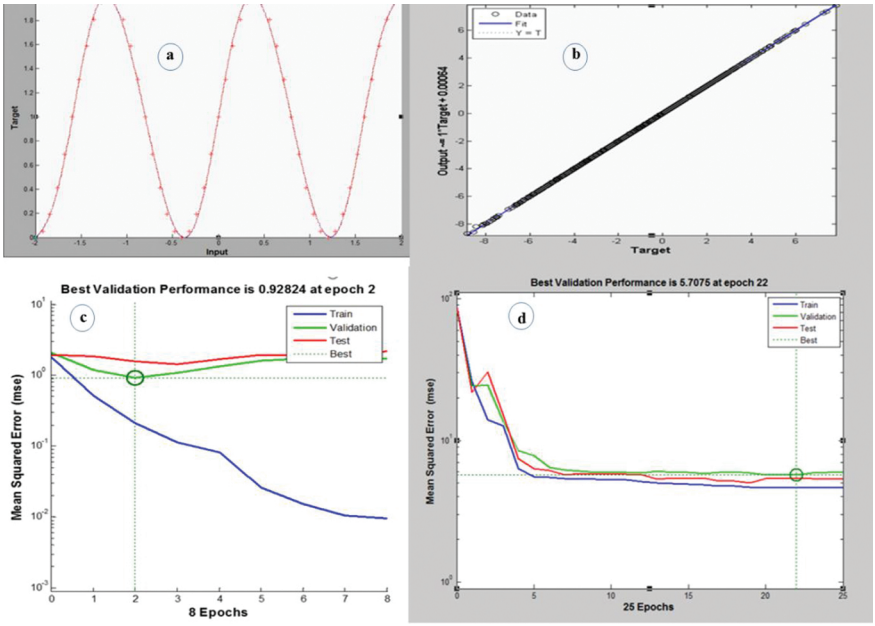


Figure 2. Function approximation of different regression models. Significant overfitting can be seen in (a)–(c).

red line in **Figure 2a**). The idealized expectation is that during training, the generalization error of the network progresses as shown in **Figure 2d**.

In general, there are two ways to deal with the overfitting problem in ANN (not considering ad hoc techniques such as pruning, greedy constructive learning, or weight sharing that reduces the number of parameters) [9]. These are **weight decay** and **early stopping**, both of which are known as *regularization* techniques. Regularization is the procedure of allowing parameter bias in the direction of what are considered to be more probable values, which reduces the variance of the estimates at the cost of introducing bias. Put in another way, regularization can be viewed as a way of compromising to minimize the objective function with regard to parameter (weights and bias) space. Next, we will briefly discuss **early stopping** before proceeding to the topic of weight decay (BR).

3.1. Early stopping

In the standard practice of backpropagation learning with early stopping, the data set is allocated into three sources: a training data set, a validation data set, and a testing data set. In most ANN practices, the biggest part of data is assigned to training data (by default a 60% of data is assigned for training in MATLAB). Each of these data sets have different task during the ANN prediction process. The training data set is used to estimate the neural network weights, while the validation data set is used to monitor the network and calculate the minimum error during the iterations till network is stopped. The last data set (test data set) is unseen data by network and task of the test data set is to decrease the bias and generate unbiased estimates for predicting future outcomes and generalizability. The test data set is used at the end of the iterative process for evaluating the performance of the model from an independently drawn sample [10]. With early stopping methods, a gradient descent algorithm is applied to the training and validation data sets. First, the training data set is used to calculate the weight and bias estimates, and then these parameter estimates are applied in the validation data set to calculate error values. The practice iterates substituting parameter estimates from the training data set into the validation data set to catch the likely smallest average error with respect to the prediction of the validation data set. Training ends when the error in the validation data set increases by certain epochs ("iterations" in statistical jargon) in order to avoid the problem of overfitting (the number of epoch is six by default in MATLAB) and then the weights at the minimum of the validation error are returned. The network parameter estimates with the best performance in the validation set are then used in analysis of the testing data to evaluate the predictive ability of the network [11].

Let the data set be $D = \{\boldsymbol{t}, (\boldsymbol{p}_i)_{i=1,\ldots,n}\}$, where $\boldsymbol{p}_i$ is a vector of inputs for individual i and $\boldsymbol{t}$ is a vector of target variables (input = independent and target = dependent variable in classical estimation terms in statistics). Once a set of weight values $\mathbf{w}$ is assigned to the connections in the networks, this defines a mapping from the input $\boldsymbol{p}_i$ to the output $\hat{t}_i$. Let M denote a specific network architecture (network architecture is the model in terms of numbers of neurons and choice of activation functions), then the typical objective function used for training a neural network using early stopping is the sum of squared estimation errors (E_D):

$$E_D(D \mid \mathbf{w}, M) = \sum_{i=1}^{n} \left(t_i - \hat{t}_i\right)^2 \tag{3}$$

for n input-target pairs defining D. Early stopping is effectively limiting the used weights in the network and thus imposes regularization, effectively lowering the Vapnik-Chervonenkis dimension. However, while early stopping often improves generalization, it does not do so in a mathematically well-defined way. It is crucial to realize that validation error is not a good estimate of generalization error. Early stopping regularization has a vital advantage over the usual regularized least square learning algorithm such as ridge regression (so-called penalized L_2) or Tikhonov regularization methods.

4. Bayesian regularization

The brief explanation given in the previous section described how early stopping works to deal with the overfitting issue in ANN. Another regularization procedure in ANN is BR, which is the linear combination of Bayesian methods and ANN to automatically determine the optimal regularization parameters (**Table 1**). In Bayesian regularized ANN (BRANN) models, regularization techniques involve imposing certain prior distributions on the model parameters. An extra term, E_w, is added by BRANN to the objective function of early stopping given in equation (3) which penalizes large weights in anticipation of reaching a better generalization and smoother mapping. As what happens in conventional backpropagation practices, a gradient-based optimization technique is then applied to minimize the function given in (4). This process is equal to a penalized log-likelihood,

$$F = \beta E_D(D|\mathbf{w}, M) + \alpha E_W(\mathbf{w}|M), \tag{4}$$

where $E_W(\mathbf{w}|M)$, is the sum of squares of architecture weights, M is the ANN architecture (model in statistical jargon), and α and β are objective function parameters (also referred to as regularization parameters or hyper-parameters and take the positive values) that need to be estimated adaptively [3]. The second term on the right hand side of equation (4), αE_W, is known as weight decay and α, the weight decay coefficient, favors small values of $\mathbf{w}$ and decreases the tendency of a model to overfit [12].

To add on a quadratic penalty function $E_W(\mathbf{w}|M)$, α yielding a version of nonlinear ridge regression with an estimate for $\mathbf{w}$ equivalent to the Bayesian maximum a prior (MAP) [1]. Therefore, the quadratic form (weight decay) favors small values of $\mathbf{w}$ and decreases the predisposition of a model to overfit the data. Here, in equation (4), training involves a tradeoff between model complexity and goodness of fit. If $\alpha >> \beta$, highlighting is on reducing the extent of weights at the expense of goodness of fit, while producing a smoother network response [13]. If Bayes estimates of α are large, the posterior densities of the weights are highly concentrated around zero, so that the weights effectively disappear and the model discounts connec-

tions in the network [12, 14]. Therefore, α and β are adaptively predicted to deal with tradeoff model complexity and goodness of fit.

MATLAB Commend		**Explanations**
load myData		Loading data
net=newff(x,y,5,{'tansig', 'purelin'},'trainbr');		Creates a new network with a dialog box. The properties of architecture created here are: tangent sigmoid (tansig) and linear activation function (purelin) in hidden and output, respectively. The number of neuron in hidden layer is 5 and the number of neuron in output layer 1 (by default). The Bayesian regularized training algorithm (trainbr) takes play for training of data.
[net,tr]=train(net,x, y)		
randn('state',192736547)		Lets you seed the uniform random number generator.
y_t=sim(net, x)		Simulate net
net = init(net)		Reinitiate net to improve results
net = train(net,x,y)		Re-train net
net.trainParam.epochs	1000	Maximum number of epochs to train
net.trainParam.goal	0	Performance goal
net.trainParam.mu	0.005	Marquardt adjustment parameter
net.trainParam.mu_dec	0.1	Decrease factor for mu
net.trainParam.mu_inc	10	Increase factor for mu
net.trainParam.mu_max	1e10	Maximum value for mu
net.trainParam.min_grad	1e-7	Minimum performance gradient
net.trainParam.show	25	Epochs between displays (NaN for no displays)
net.trainParam. showCommandLine	False	Generate command-line output
net.trainParam. showWindow	True	Show training GUI

Table 1. Training occurs according to Bayesian regularization algorithms (trainbr) training parameters, shown here with their default values (commends given in bold face are required).

As stated in the early stopping regularization section of this chapter, the input and target data set is typically divided into three parts; the training data set, the validation data set, and the test data sets. In BRANNs, particularly when the input and target data set is small, it is not essential to divide the data into three subsets: training, validation, and testing sets. Conversely all available data set is devoted to model fitting and model comparison [15]. This implementation is important when training networks with small data sets as is thought that BR has better generalization ability than early stopping (http://www.faqs.org/faqs/ai-faq/neural-nets/part3/section-5.html).

The strength connections, **w**, of the network are considered random variables and have no meaning before training. After the data is taken, the density function for the weights can be updated according to Bayes' rule. The empirical Bayes approach in [12] is as follows. The posterior distribution of **w** given α, β, D, and M is

$$P(\mathbf{w} \mid D,\alpha,\beta,M) = \frac{P(D|\mathbf{w},\beta,M)P(\mathbf{w}|\alpha,M)}{P(D \mid \alpha,\beta,M)}, \tag{5}$$

where D is the training data set and M is the specific functional form of the neural network architecture considered. The other terms in equation (5) are:

- $P(\boldsymbol{w} \mid D, \alpha, \beta, M)$ is the posterior probability of **w**,
- $P(D \mid \boldsymbol{w}, \beta, M)$ is the likelihood function of **w**,
- $P(\boldsymbol{w} \mid \alpha, M)$ is the prior distribution of weights under M, which is the probability of observing the data given **w** and
- $P(D \mid \alpha, \beta, M)$ is a normalization factor or evidence for hyperparameters α and β.

The normalization factor does not depend on **w** (Kumar, 2004). That is,

$$P(D \mid \alpha,\beta,M) = \int P(D|\mathbf{w},\beta,M)P(\mathbf{w}|\alpha,M)d\mathbf{w}.$$

The weights **w**, were assumed to be identically distributed, each following the Gaussian distribution ($\mathbf{w} \mid \alpha, M) \sim N(0, \alpha^{-1}$). Given this, the expression of joint prior density of **w** in equation (5) is

$$p(\mathbf{w} \mid a,M) \propto \prod_{l=1}^{m} \exp\left(-\frac{\alpha w_{kj}^2}{2}\right) = \exp\left[-\frac{\alpha E_W(\mathbf{w} \mid M)}{2}\right].$$

After normalization, the prior distribution is then [2]

$$p(\mathbf{w} \mid \alpha,M) = \frac{\exp\left[-\frac{\alpha E_W(\mathbf{w} \mid M)}{2}\right]}{\int \exp\left[-\frac{\alpha E_W(\mathbf{w} \mid M)}{2}\right] d\mathbf{w}} = \frac{\exp\left[-\frac{\alpha E_W(\mathbf{w} \mid M)}{2}\right]}{Z_w(\alpha)}, \tag{6}$$

where $Z_W(\alpha) = \left(\frac{2\pi}{\alpha}\right)^{\frac{m}{2}}$.

http://dx.doi.org/10.5772/63256

As target variable t, is a function of input variables, $\mathbf{p}$, (the same association between dependent and independent variables in regression model) it is modeled as $t_i = f(\mathbf{p}_i) + e$, where $e \sim N(0, \beta^{-1})$ and $f(\mathbf{p}_i)$ is the function approximation to $E(t \mid \mathbf{p})$. Assuming Gaussian distribution, the joint density function of the target variables given the input variables, β and M is [2]:

$$\begin{aligned} P(\mathbf{t} \mid \mathbf{p}, \mathbf{w}, \beta, M) &= \left(\frac{\beta}{2\pi}\right)^{\frac{N}{2}} \exp\left[-\frac{\beta}{2}\sum_{i=1}^{N}(t_i - f(\mathbf{p_i}))^2\right] \\ &= \left(\frac{\beta}{2\pi}\right)^{\frac{N}{2}} \exp\left[-\frac{\beta}{2}E_D(D \mid \mathbf{w}, M)\right] \end{aligned} \tag{7}$$

where $E_D(D \mid \mathbf{w}, M)$ is as given in equation (3) and (4). Letting $Z_D(\beta) = \int \exp\left[-\frac{\beta}{2}E_D(D \mid \mathbf{w}, M)\right] = \left(\frac{2\pi}{\beta}\right)^{\frac{N}{2}}$, the posterior density of $\mathbf{w}$ in equation (5) can be expressed as

$$\begin{aligned} P(\mathbf{w} \mid D, \alpha, \beta, M) &= \frac{\frac{1}{Z_W(\alpha)Z_D(\beta)}\exp\left[-\frac{1}{2}(\beta E_D + \alpha E_W)\right]}{P(D \mid \alpha\ \beta\ M)}, \\ &= \frac{1}{Z_F(\alpha, \beta)}\exp\left[-\frac{F(w)}{2}\right] \end{aligned} \tag{8}$$

where $Z_F(\alpha, \beta) = \left[Z_W(\alpha)Z_D(\beta)P(D \mid \alpha, \beta, M\right]$ and $F = \beta E_D + \alpha E_W$. In an empirical Bayesian framework, the "optimal" weights are those that maximize the posterior density $P(\mathbf{w} \mid D, \alpha, \beta, M)$. Maximizing the posterior density of $P(\mathbf{w} \mid D, \alpha, \beta, M)$ is equivalent to minimizing the regularized objective function F given in equation (4). Therefore, this indicates that values of α and β need to be predicted by architecture M, which will be further discussed in the next section.

While minimization of objective function F is identical to finding the (locally) maximum *a posteriori* estimates $\mathbf{w}^{MAP}$, minimization of E_D in F by any backpropagation training algorithm is identical to finding the maximum likelihood estimates $\mathbf{w}^{ML}$ [12]. Bayesian optimization of the regularization parameters require computation of the Hessian matrix of the objective function F evaluated at the optimum point $\mathbf{w}^{MAP}$ [3]. However, directly computing the Hessian matrix is not always required. As proposed by MacKay [12], the Gauss-Newton approximation to the Hessian matrix can be used if the LM optimization algorithm is employed to locate the minimum of F [16].

4.1. Brief discussion of Levenberg-Marquardt optimization

The LM algorithm is a robust numerical optimization technique for mapping as well as function approximation. The LM modification to Gauss-Newton is

$$(\mu \mathbf{I}) \boldsymbol{\delta} = \mathbf{J}' \mathbf{e} \tag{9}$$

and the Hessian matrix can be approximated as:

$$\mathbf{H} = \mathbf{J}' \mathbf{J}, \tag{10}$$

where **J** is the Jacobian matrix that contains first derivatives of the network errors with respect to network parameters (the weights and biases), μ is the Levenberg`s damping factor, and $\boldsymbol{\delta}$ is the parameter update vector. The δ indicates how much the magnitude of weights needs to be changed to attain better prediction ability. The way to calculate **J** matrix is given in equation (11). The gradient of the ANN is computed as **g=J'e**. The μ in equation (9) is adjustable in each iteration, and guides the optimization process during ANN learning with the training data set. If reductions of the cost function F are rapid, then the parameter μ is divided by a constant (c) to bring the algorithm closer to the Gauss-Newton, whereas if an iteration gives insufficient reduction in F, then μ is multiplied by the same constant giving a step closer to the gradient descent direction (http://crsouza-blog.azurewebsites.net/2009/11/neural-network-learning-by-the-levenberg-marquardt-algorithm-with-bayesian-regularization-part-1/#levenberg). Therefore, the LM algorithm can be considered a trust-region modification to Gauss-Newton designed to serve as an intermediate optimization algorithm between the Gauss-Newton method and the Gradient-Descent algorithm [17].

The Jacobian matrix is a matrix of all first-order partial derivatives of a vector-valued function. The dimensions of the matrix are formed by the number of observations in the training data and the total number of parameters (weights + biases) in the ANN being used. It can be created by taking the partial derivatives of each output with respect to each weight, and has the form:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial e_1(w)}{\partial w_1} & \frac{\partial e_1(w)}{\partial w_2} & \cdots & \frac{\partial e_1(w)}{\partial w_n} \\ \frac{\partial e_2(w)}{\partial w_1} & \frac{\partial e_2(w)}{\partial w_2} & \cdots & \frac{\partial e_2(w)}{\partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_N(w)}{\partial w_1} & \frac{\partial e_N(w)}{\partial w_2} & \cdots & \frac{\partial e_N(w)}{\partial w_n} \end{bmatrix} \tag{11}$$

The parameters at l iteration are updated as in equation (12) when the Gauss-Newton approximation of the Hessian matrix by LM in Bayesian regularized neural network (BRANN) is used.

http://dx.doi.org/10.5772/63256

$$w^{l+1} = w^l - \left[\mathbf{J}^T\mathbf{J} + \mu\,\mathbf{I}\right]^{-1}\mathbf{J}^T\mathbf{e} \tag{12}$$

Therefore, this approach provides a numerical solution to the problem of minimizing a nonlinear function over the parameter space. In addition, this approach is a popular alternative to the Gauss-Newton method of finding the minimum of a function (http://crsouza-blog.azurewebsites.net/2009/11/neural-network-learning-by-the-levenberg-marquardt-algorithm-with-bayesian-regularization-part-1/#levenberg). Next, let's compare the equations for some common training algorithms and use this information to decipher how parameters are updated after each iteration (epoch) in ANN: Updating the strength connection in standard backpropagation training $w^{l+1} = w^l - \alpha\frac{\partial E_D}{w^l}$,

updating the strength connection in Quasi-Newton training $w^{l+1} = w^l - \mathbf{H}_l^{-1}\alpha\frac{\partial E}{w^l}$, and updating the strength connection in LM training $w^{l+1} = w^l - \left[\mathbf{J}^T\mathbf{J} + \mu\,\mathbf{I}\right]^{-1}\mathbf{J}^T\mathbf{e}$.

Therefore, Quasi-Newton is necessary for actual calculation of the Hessian matrix **H** while LM is used to approximate **H**. The α coefficient in the first two equations is referred to as the learning rate. The performance of the algorithm is very sensitive to proper setting of the learning rate. If the learning rate is set too high, the algorithm can oscillate and become unstable. On the other hand, too low learning rate makes the network learn very slowly and converging of ANN will take a while. In gradient descent learning, it is not practical to determine the optimal setting for the learning rate before training, and in fact, the optimal learning rate changes during the training process, as the algorithm moves across the performance surface [18].

5. Tuning parameters α and β

As discussed earlier, minimizing the regularized objective function $F = \beta E_D(D\,|\,\boldsymbol{w},M) + \alpha E_W(\boldsymbol{w}\,|\,M)$ in equation (4) is equivalent to maximization of the posterior density $P(\mathbf{w}\,|\,D,\,\alpha,\,\beta,\,M)$. A typical procedure used in neural networks infers α and β by maximizing the marginal likelihood of the data in equation (5). The joint probability of α and β is

$$P(\alpha,\beta\,|\,D,M) = \frac{P(D|\alpha,\beta,M)P(\alpha,\beta|M)}{P(D\,|\,M)}. \tag{13}$$

Essentially, we need to maximize the posterior probability $P(\alpha\ \beta \mid D, M)$ with respect to hyperparameters α and β which is equivalent to maximization of $P(D \mid \alpha, \beta, M)$. $P(D \mid \alpha, \beta, M)$ is the normalization factor given for the posterior distribution of $\mathbf{w}$ in equation (5).

If we apply the unity activation function in equation (5), which is the analog of linear regression, then this equation will have a closed form. Otherwise, equation (5) does not have a closed form if we apply any sigmoidal activation function, such as logit or tangent hyperbolic in the hidden layer. Hence, the marginal likelihood is approximated using a Laplacian integration completed in the area of the current value $\mathbf{w}=\mathbf{w}^{\mathrm{MAP}}$ [3]. The Laplacian approximation to the marginal density in equation (5) is expressed as

$$\log\left[p(D \mid \alpha,\beta,M)\right] \approx K + \frac{n}{2}\log(\beta) + \frac{m}{2}\log(\alpha) - \left|F(\alpha,\beta)\right|_{w=w^{MAP}_{(\alpha,\beta)}} - \frac{1}{2}\log \| \mathbf{H} \|_{w=w^{MAP}_{(\alpha,\beta)}} \tag{14}$$

where K is a constant and $\mathbf{H} = \frac{\partial^2}{\partial \mathbf{w} \partial \mathbf{w}'} F(\alpha,\beta)$ is the Hessian matrix as given in equation (10) and (11). A grid search can be used to locate the α, β maximizers of the marginal likelihood in the training set. An alternative approach [12, 14] involves iteration (updating is from right to left, with $\mathbf{w}^{\mathrm{MAP}}$ evaluated at the "old" values of the tuning parameters)

$$\alpha_{new} = \frac{m}{2\left(\mathbf{w}^{MAP}{}'\mathbf{w}^{MAP} + tr\mathbf{H}^{-1}_{MAP}\right)}$$

and

$$\beta_{new} = \frac{n - m + 2\alpha_{MAP} tr\mathbf{H}^{-1}_{MAP}}{2\sum_{i=1}^{n}\left(t_i - b - \sum_{k=1}^{S} w_k g_k\left(b_k + \sum_{j=1}^{n} a_{ij} u_j^{*[k]}\right)\right)^2_{w=w^{MAP}_{(\alpha,\beta)}}} \quad \Rightarrow \beta_{new} = \frac{n-\gamma}{2E_D(\mathbf{w}^{MAP})} \tag{15}$$

The expression $\gamma = m - 2\alpha_{new} tr(\mathbf{H}^{MAP})^{-1}$ is referred to as the number of effective parameters in the neural network and its value ranges from 0 (or 1, if an overall intercept b is fitted) to m, the total number of connection strength coefficients, w, and bias, b, parameters in the network. The effective number of parameters indicates the number of effective weights being used by the network. Subsequently, the number of effective parameters is used to evaluate the model complexity and performance of BRANNs. If γ is close to m, over-fitting results, leading to poor generalization.

http://dx.doi.org/10.5772/63256

6. Steps in BRANN

A summary of steps in BRANN is given in **Figure 3**. Initialize α, β and the weights, w. After the first training step, the objective function parameters will recover from the initial setting.

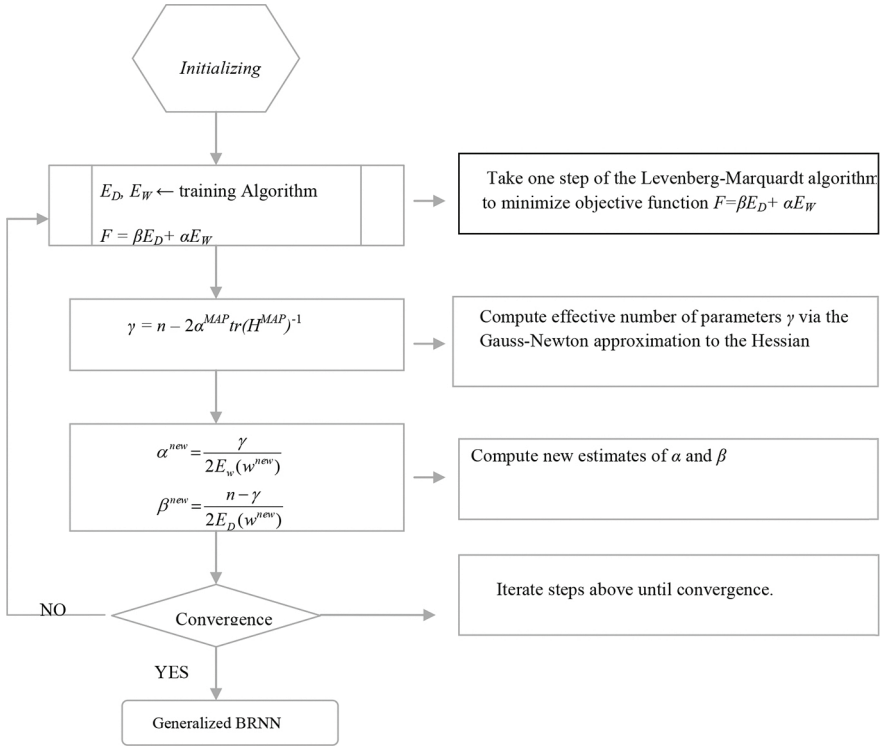


Figure 3. Flow chart for Bayesian optimization of regularization parameters α and β in the neural networks; MAP=maximum *a posteriori* [partially adapted from 16 and 2].

1. Take one step of the LM algorithm to minimize the objective function $F(\alpha,\beta)$ and find the current value of **w**.

   i. Compute the Jacobian as given in equation (11).

   ii. Compute the error gradient $\boldsymbol{g} = \mathbf{J}^T\mathbf{e}$.

   iii. Approximate the Hessian matrix using $\mathbf{H} = \mathbf{J}^T\mathbf{J}$.

   Calculate the objective function as given in equation (4).

   Solve the $\mathbf{J'J} + \mu\mathbf{I})\boldsymbol{\delta} = \mathbf{J'e}$.

Update the network weights **w** using **δ**.

2. Compute the effective number of parameters $\gamma = \boldsymbol{N} - 2atr(\mathbf{H})^{-1}$ making use of the Gauss-Newton approximation to the Hessian matrix available from the LM training algorithm.
3. Compute updated α_{new} and β_{new} as
   $\alpha^{MAP} = \frac{\gamma}{2E_w(w^{MAP})}$ and $\beta^{MAP} = \frac{m-\gamma}{2E_D(w^{MAP})}$; and
4. Iterate steps 2-4 until convergence.

7. Practical considerations

In general, there are three steps between input and predicted values in ANN training. These are pre-processing, network object (training), and post-processing. These steps are important to make ANN more efficient as well as drive optimal knowledge from ANN. Some of these steps will be considered in the following sections.

7.1. Data cleaning

Before training, the data sets should be checked in terms of constant, corrupt, and incorrect values from the input and target data set. After checking for suspicious and improper data, the dimension of the input vector could be an important issue for efficient training. In some situations, the dimensions of the data vectors could be large, but the components in the data vectors can be highly correlated (redundant). Principal component analysis (PCA) is the most common technique for dimensionality reduction and orthogonalization.

7.2. Normalization

Normalization or scaling is not really a functional requirement for the neural networks to learn, but it significantly helps as it converts the input and target (independent and dependent variables in statistical jargon) into the data range that the sigmoid activation functions lie in (i.e. for logistic [0, 1] and tanh [−1, 1]. For example, in MLP, non-linear functions in hidden layers become saturated when the input is larger than six (exp(-6)~0.00247). Consequently, large inputs cause ill-conditioning by leading to very small weights. Further with large inputs, the gradient values become very small, and the network training will be very slow.

By default, before initiating the network processing, MATLAB Neural Network Toolbox® rescales both input and output variables such that they lie -1 to +1 range, to boost numerical stability. This task is done spontaneously in MATLAB Neural Network Toolbox® using the "mapminmax" function. To explain this, consider the simple data vector as x'=[8, 1, 5]. Here the x_{min} = 1 and x_{max} = 8. If values are to range between A_{min} = -1 and A_{max} = +1, one redefine the

data vector temporarily as x_{temp}'= [−1, 1, 5], so only x_3= 5 needs to be rescaled to guarantee that all variables reside between -1 and +1. This is done by using the following formula:

$$x_{3,new} = A_{\min} + \frac{x_3 - x_{\min}}{x_{\max} - x_{\min}}(A_{\max} - A_{\min}) = -1 + \frac{5-1}{8-1}2 = -0.143.$$

When the input and target data is normalized, the network output always falls into a normalized range. The network output can then be reverse transformed into the units of the original target data when the network is put to use in the field [18].

7.3. Multiple runs

Each time a neural network is trained, the output obtained from each run can result in a different solution mainly due to: a) different initial weight and bias values, and b) different splitting of data into training, validation, and test sets (or into training and testing). As a result, function approximation using different neural network architectures trained on the same problem can produce different outputs for the same input fed into the ANN. For example, depending on the initial weights of the network, the algorithm may converge to local minima or not converge at all. To avoid local minima convergence and overtraining, improve the predictive ability of ANN, and eliminate spurious effects caused by random starting values. Several independent BRANNs, say 10, should be trained for each architecture. Results are then recorded as the average of several runs on each architecture.

7.4. Analyses and computing environment

Determination of the optimal number of neurons in the hidden layer is an essential task in ANN architectures. As stated earlier, network with only a few neurons in hidden layer may be incapable of capturing complex association between target and input variables. However, if too excessive neurons are assigned in hidden part of the network then it will follow the noise in the data due to overparameterization, leading to bad generalization and poor predictive ability of unseen data [3, 19]. Therefore, a different number of neurons in the hidden layer should be tried, and architecture performance should be assessed after each run with a certain number of neurons in the hidden layer. Because highly parameterized models are penalized in BRANN, the complexity of BRANN architectures using sum of squares weights as well as the degree of shrinkage attained by BRANN can be used to determine the optimal number of neurons. The number of parameters (weights and biases) is a function of the number of neurons. More neurons in the hidden layer imply that more parameters need to be estimated. Therefore, one criteria for model selection in complex BRANN concerns the number of weights. The more weights there are, relative to the number of training cases, the more overfitting amplifies noise in the target variables [1–3]. As demonstrated in **Figure 4**, BRANN also calculates the effective number of parameters to evaluate the degree of complexity in ANN architecture. BRANN uses the LM algorithm (based on linearization) for computing the

posterior modes in BR. The penalized residual sum of squares is also used to determine the number of neurons in the hidden part of the network.

7.5. Stopping training of BRANN

In MATLAB applications, training of BRANN is stopped if:

1. The maximum number of epochs (number of iterations in statistical jargon) is reached
2. Performance of the network with the number of neurons and combinations of the activation functions has met a suitable level
3. The gradient (**J'e**) was below a suitable target
4. The LM μ parameter exceeded a suitable maximum (training stopped when it became larger than 10^{10}).

Each of these targets and goals are set as the default values in MATLAB implementation. The maximum number of iterations (called epochs) in back-propagation was set to 1000, and iteration will stop earlier if the gradient of the objective function is below a suitable level, or when there are obvious problems with the algorithm [20].

8. Interpreting the results

8.1. Degree of complexity

This example illustrates the impact of shrinkage and how regularized neural networks deal with the "curse of dimensionality". The example given here has 500 inputs and the number of neurons in the hidden layer is 5, such that the total of nominal parameters (weights and biases) is 2511. However, the effective number of parameters is 316 (**Figure 4**). In other words, the model can be explained well with only with 316 parameters when BR is used for training the networks. In practice, different architectures (different activation functions and different numbers of neurons in the hidden layer) should be explored to decide which architecture best fits the data. Hence, incremental increasing of the number of neurons from one to several is the best practice to cope with the "curse of dimensionality".

8.2. Predictive performance

The predictive correlation is 0.87 (**Figure 4**) in the given example, which is quite high, implying that the predictive ability of the model used is sufficient. However, as stated earlier, to eliminate spurious effects caused by random starting values, several independent BRANNs should be trained for each architecture and the average value of multiple runs should be reported.

http://dx.doi.org/10.5772/63256

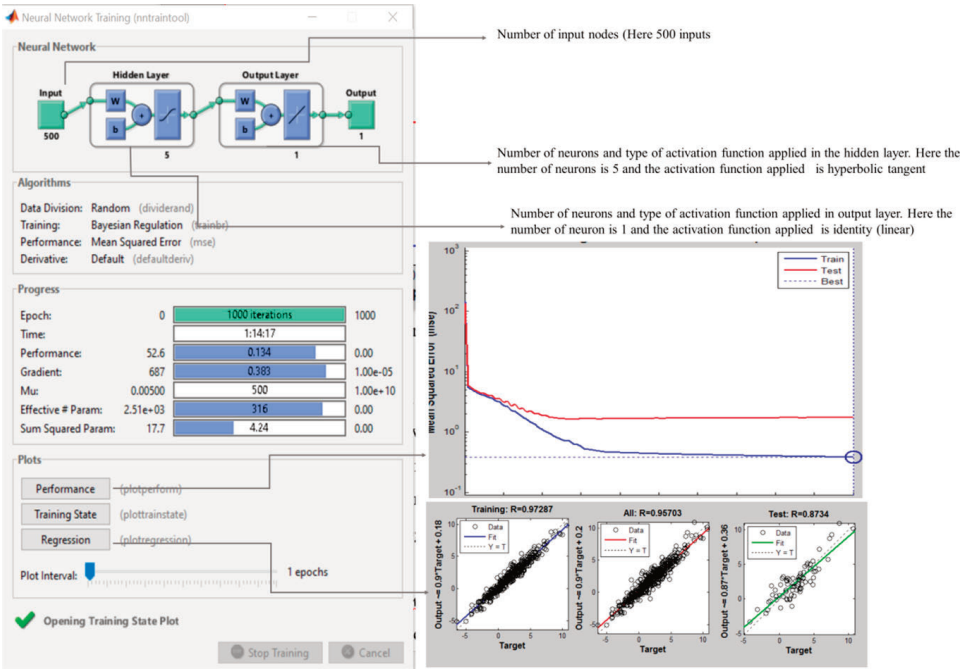


Figure 4. Output screens from MATLAB runs.

8.3. Shrinkage

The distribution of connection strengths (network parameters) in an ANN gives an evidence of the degree of regularization achieved by BRANNs. Conventionally, weight values are shrinkage with model complexity, in the same approach that estimates of network parameters become smaller in BR implementations. This is true when inputs p increase and training sample size remains constant. Moreover, the distribution of weights in any architecture is often associated with the predictive ability of network; small value weights tend to lead to better generalization for unseen data [1, 19]. **Figure 4** shows the distributions of weights for the nonlinear regularized networks with five neurons in the hidden layer. It is suggested that a linear activation function in the hidden and output layers, as well as a nonlinear activation function with different numbers of neurons in the hidden and a linear activation function in output layer should be tried. This provides a good opportunity to compare models for the extent of regularization attained. For example, the sum of squared weights for about 2500 parameters is only 4.24 in the given example (see **Figure 4**), indicating how Bayesian neural networks reduce the effective number of weights relative to what would be obtained without regularization. In other words, the

method by which highly parameterized models are penalized in the Bayesian approach helps to prevent over-fitting.

9. Conclusion

Small *n and big p* (*p>>n*) is key issue for prediction of complex traits from big data sets, and ANNs provide efficient and functional approaches to deal with this problem. Because the competency of capturing highly nonlinear relationship between input and outcome variables, ANNs act as universal approximators to learn complex functional forms adaptively by using different type of nonlinear functions. Overfitting and over-parameterization are critical concerns when ANN is used for prediction in (*p>>n*). However, BRANN plays a fundamental role in attenuating these concerns. The objective function used in BRANN has an additional term that penalizes large weights to attain a smoother mapping and handle overfitting problem. Because of the shrinkage, the effective number of parameters attained by BRANN is less than the total number of parameters used in the model. Thus, the over-fitting is attenuated and generalization ability of the model is improved considerably.

Acknowledgements

Author would like to sincerely thank Jacob Keaton for his reduction, and comments.

Author details

Hayrettin Okut

Address all correspondence to: okut.hayrettin@gmail.com

Yüzüncü Yıl University, Faculty of Agriculture, Biometry and Genetic Branch, Van, Turkey

Wake Forest University, School of Medicine, Center for Diabetes Research, Center for Genomics and Personalized Medicine Research, Winston-Salem, NC, USA

References

[1] Gianola, D., Okut, H., Weigel, K., Rosa, J. G. Predicting complex quantitative traits with Bayesian neural networks: a case study with Jersey cows and wheat. BMC Genetics. 2011. 12:87.

[2] Okut, H., Gianola, D., Rosa, J. G., Weigel, K. Prediction of body mass index in mice using dense molecular markers and a regularized neural network. Genetics Research (Cambridge). 2011. 93:189–201

[3] Okut, H., Wu, X. L., Rosa, J. M. G., Bauck, S., Woodward, B., Schnabel, D. R., Taylor, F. J., Gianola, D. Predicting expected progeny difference for marbling score in Angus cattle using artificial neural networks and Bayesian regression models. Genetics Selection Evolution. 2014. 45:34. DOI: 10.1186/1297-9686-45-34.

[4] Lampinen, J., Vehtari, A. Bayesian approach for neural networks review and case studies. Neural Networks. 2001. 14:257–274.

[5] SAS® Enterprise Miner™ 14.1. Administration and Configuration Copyright©. Cary, NC: SAS Institute Inc.; 2015.

[6] Feng, N., Wang, F., Qiu, Y. Novel approach for promoting the generalization ability of neural networks. International Journal of Signal Processing. 2006. 2:131–135.

[7] Ping, G., Michael, R. L., Chen, C. L. P. Regularization parameter estimation for feedforward neural networks. IEEE Transactions on Systems, Man, and Cybernetics—PART B: Cybernetics. 2003. 33:35–44.

[8] Wang, H. J., Ji, F., Leung, C. S., Sum, P. F. Regularization parameter selection for faulty neural networks. International Journal of Intelligent Systems and Technologies. 2009. 4:45–48.

[9] Marwala, T. Bayesian training of neural networks using genetic programming. Pattern Recognition Letters. 2007. 28:1452–1458.

[10] Matignon, R. Data Mining Using SAS Enterprise Miner. Chicago: Wiley; 2007. p. 584. ISBN: 978-0-470-14901-0. DOI: 10.1002/9780470171431

[11] Okut, H., Gianola, D., Rosa, J. G., Weigel, K. Evaluation of prediction ability of Cholesky factorization of genetic relationship matrix for additive and non-additive genetic effect using Bayesian regularized neural network. IORE Journal of Genetics. 2015. 1(1). 1–15

[12] MacKay, J. C. D. Information Theory, Inference and Learning Algorithms. Cambridge University Press, Cambridge-UK; 2008.

[13] Foresee, F. D., Hagan, M. T. Gauss-Newton approximation to Bayesian learning. Proceedings: IEEE International Conference on Neural Networks. 1997. Vol:3, 1930–1935.

[14] Titterington, D. M. Bayesian methods for neural networks and related models. Statistical Science. 2004. 19:128–139.

[15] Bishop, C. M., Tipping, M. E. A hierarchical latent variable model for data visualization. IEEE Transactions on Pattern Analysis and Machine Intelligence. 1998. 20(3):281–293.

[16] Shaneh, A., Butler, G. Bayesian learning for feedforward neural network with application to proteomic data: the glycosylation sites detection of the epidermal growth factor-

like proteins associated with cancer as a case study. In Canadian AI LNAI 4013, 2006 (ed. L. Lamontagne & M. Marchand). Berlin-Heiddelberg: Springer-Verleg; 2006. 110–121.

[17] Battiti, R. Using mutual information for selecting features in supervised neural net learning neural networks. IEEE Transactions on Neural Networks. 1994. 5(4):537–550.

[18] Demuth, H., Beale, M., Hagan, M. Neural Network ToolboxTM 6 User's Guide. Natick, MA: The MathWorks Inc.; 2010.

[19] Felipe, P.S. V., Okut, H., Gianola, D., Silva, A. M., Rosa, J. M. G. Effect of genotype imputation on genome-enabled prediction of complex traits: an empirical study with mice data. BMC Genetics. 2014. 15:149. DOI: 10.1186/s12863-014-0149-9

[20] Haykin, S. Neural Networks: Comprehensive Foundation. New York: Prentice-Hall; 2008.