

**U. S. Department of Commerce
National Oceanic and Atmospheric Administration
National Weather Service
National Centers for Environmental Prediction**

Office Note 478

**NN-TVS,
NCEP Neural Network Training and
Validation System**

Brief Description of NN Background and Training Software

Vladimir M. Krasnopolsky

Environmental Modeling Center, NCEP/NWS, NOAA

October 2014

MMAB Contribution No. 320

e-mail: Vladimir.Krasnopolsky@NOAA.gov

Table of Contents

- Abstract.....4
- 1 Introduction5
- 2 Using NN to Emulate Mappings: Basic Ideas7
 - 2.1 Mappings and NNs, Background 7
 - 2.1.1 Mapping dimensionalities, domain, and range7
 - 2.1.2 Multilayer Perceptron NNs9
 - 2.1.3 NNs in their traditional terms10
 - 2.2 Preparation for NN Training 12
 - 2.2.1 Training set12
 - 2.2.2 Selection of the NN architecture14
 - 2.2.3 Normalization of the NN inputs and outputs16
 - 2.2.4 Constant inputs and outputs18
 - 2.3 NN training..... 18
 - 2.3.1 Batch training and sequential training20
 - 2.3.2 Missed inputs and outputs.....21
 - 2.3.3 Overfitting and regularization23
 - 2.4 Advantages and Limitations of the NN Technique 24
 - 2.4.1 Flexibility of the MLP NN.....24
 - 2.4.2 NN training, nonlinear optimization, and correlation of inputs and outputs25
 - 2.4.3 NN generalization: interpolation and extrapolation26
 - 2.4.4 NN Jacobian.....26
 - 2.4.5 Multiple NN emulations for the same target mapping and NN ensemble approaches.....29
 - 2.4.6 Estimates of NN parameters uncertainty30
 - 2.4.7 NNs vs. physically based models; NN as a “black box”30
 - 2.5 NN Emulations 32
 - 2.6 Final remarks 33
 - 2.7 References 34
- 3 The NN-TVS structure and data sets38
 - 3.1 The structure of the NN-TVS system 38
 - 3.2 Starting the system 39
 - 3.3 Running the main program *nntrain*, the structure of training sets 44
 - 3.4 Running the program *pnetrain* 45

3.5	Restarting NN training	49
3.6	Running the FORTRAN code <i>trainl</i>	49
3.7	Returning to the IDL codes <i>pnetrain</i> and <i>nntrain</i>	51
4	Using trained NN	52
5	Conclusions	53
6	Appendixes	54
	Appendix 1. Data files used by NN-TVS	54
	Appendix 2. List of IDL and FORTRAN functions and subroutines used by NN-TVS	55
	Appendix 3. FORTRAN module <i>neural.f90</i> , which implements trained NN...	56
	Appendix 4. Output Statistics calculated by NN-TVS	59

Abstract

This Note describes neural network (NN) training and validation system (TVS) or NN-TVS developed at NCEP (EMC). Section 1 of this Note is an introductory section. Section 2 presents a brief NN tutorial containing limited discussion of basic ideas of NN technique and NN features implemented in the NN-TVS system. Some of the NN features discussed in Section 2 have not yet been implemented in the NN-TVS system and are expected to be implemented in following versions of the system. Section 3 contains a description of the NN-TVS structure and functional relationships between blocks. It also describes scripts necessary for running the system on NCEP computers. Section 4 contains a description of FORTRAN module neural.f90 that can be used to run the trained NN in applications. Section 5 contains appendixes with information about the data files used in NN-TVS (Appendix 1); also the functions and subroutines constituting the NN-TVS system are listed in Appendix 2. The text of the FORTRAN module neural.f90 is presented in Appendix 3. Appendix 4 explains the evaluation statistics that are used in the NN-TVS evaluations of NN training results.

1 Introduction

This Note describes a neural network (NN) training and validation system (TVS) or NN-TVS developed at NCEP (EMC) during last two decades. This system evolved significantly during this time following the evolution of problems to solve. Almost all NN application developed during this period of time at NCEP, as well as majority of oceanic and atmospheric applications developed elsewhere, from the mathematical point of view, can be formulated as complex, multidimensional, nonlinear mappings (see Krasnopolsky 2007, 2013 for examples). Accordingly, a particular type of NN – the Multilayer Perceptron (MLP) (Rumelhart et al. 1986) has been utilized in the NN-TVS system because MLP is a universal approximator capable of approximating any continuous mapping (Hornik et al. 1990).

A mapping, M , between two vectors X (input vector) and Y (output vector) can be symbolically written as,

$$Y = M(X); \quad X \in \mathfrak{R}^n, Y \in \mathfrak{R}^m \quad (2.1)$$

where n and m are the dimensionalities of the input and output spaces correspondingly. A large number of important practical applications may be considered mathematically as a mapping (2.1). Keeping in mind that a NN technique will be used to approximate (or emulate) this mapping, we will call this mapping a *target mapping*, using a term taken from nonlinear approximation theory (DeVore 1998).

The target mapping may be defined explicitly or implicitly. It can be defined explicitly as a set of equations based on first principles and/or empirical dependencies (e.g., radiative transfer or heat transfer equations) or as a computer code. A collection of data records (e.g., observations, measurements, computer simulations) represents the target mapping implicitly. The target mapping is assumed to represent these data and to generate them as well.

Section 2 of this Note presents a brief NN tutorial containing limited discussion of basic ideas of NN technique and NN features implemented (or to be implemented) in the NN-TVS system. Some of the NN features discussed in Section 2 have not yet been implemented in the NN-TVS

system and expected to be implemented in following versions of the system. Readers interested in only in description of the NN-TVS and running the system can skip Section 2.

Section 3 contains a description of the NN-TVS structure and functional relationships between blocks. It also describes scripts necessary for running the system on NCEP computers and data sets that the system uses. Section 4 contains a description of FORTRAN module neural.f90 that can be used to run the trained NN in applications. Section 5 contains appendixes with information about the data files used in NN-TVS (Appendix 1); also the functions and subroutines constituting the NN-TVS system are listed in Appendix 2. The text of the FORTRAN module neural.f90 is presented in Appendix 3.

2 Using NN to Emulate Mappings: Basic Ideas

Many books have been devoted to introducing NN technique and its applications. The basic ideas are well presented by Beale and Jackson (1990). More advanced information about the NN theory and properties can be found in several books by: Bishop (1995 and 2006), Cherkassky and Mulier (2007), and Haykin (2008). Atmospheric, oceanic, and other environmental and Earth system applications are introduced in several review papers and books by: Krasnopolsky (2007), Haupt, Pasini, and Marzban (2009), Hsieh (2009), and Krasnopolsky (2013). In this section, only basic ideas necessary for understanding the NN technique implemented (or to be implemented) in the software system NN-TVS and for practical use of this technique and software are presented following Krasnopolsky (2013).

2.1 Mappings and NNs, Background

2.1.1 Mapping dimensionalities, domain, and range

The first essential property of the target mapping is its *mapping dimensionalities*. A mapping is characterized by two dimensionalities: (i) the dimensionality n of the input space, \mathfrak{R}^n , and (ii) the dimensionality m of the output space, \mathfrak{R}^m . Here we consider only mappings between vectors X and Y , which both consist of real numbers as their components. In this case, both input and output spaces are real vector spaces.

The second important property of the mapping (2.1) is the *mapping domain*. Only a part of the input space \mathfrak{R}^n is spanned by the input vectors X . This part is called the mapping domain, D , and is determined by the particular application. Understanding the configuration of the mapping domain and its properties is essential for any application of the mapping (2.1) and for proper NN training and use (Bishop 1995). If all components of the input vector X are scaled to the range $[-1, 1]$, the volume of the input space \mathfrak{R}^n is equal to 2^n and, therefore, grows exponentially with n . Once the space is discretized by K values per dimension, then the problem grows even faster, as K^n . It means that in the input space we have K^n grid cells, and to represent our mapping, we need an exponentially large training set in order to ensure that each grid cell contains at least one

data point. This exponential growth in the amount of data with the increase of the input space dimensionality is often called the *curse of dimensionality* (Bishop 1995, Vapnik and Kotz 2006).

Fortunately, in a particular application, the components of the input vector, X , are usually inter-related or multi-collinear (Aires et al. 2004b) due to physical or statistical constraints that lead to both positive and negative consequences. On the positive side, these correlations effectively reduce the size, and sometimes dimensionality, of the part of the input space \mathfrak{R}^n spanned by the input vectors X (the mapping domain, D). As a result, for a particular application, the mapping domain, D , may be significantly smaller than an n -dimensional cube in the input space \mathfrak{R}^n . On the negative side, it is often very difficult, if possible at all, to determine the actual shape and effective dimensionality of D , which makes it difficult to adequately sample the mapping domain D and to select the optimal architecture of the NN that emulates the target mapping.

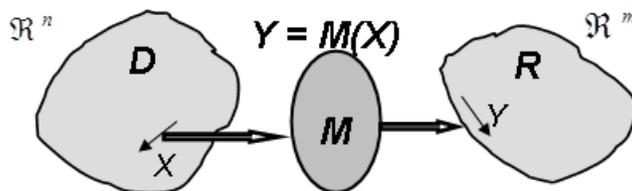


Figure 2.1. The mapping (2.1), M , its input vector, X , output vector, Y , domain, D , and range, R .

The components of the output vector, Y , are also usually inter-related. As a result, the output vectors also span only a fraction of the output space \mathfrak{R}^m . This part of the output space is called the *range*, R . Understanding the properties of the range is very important for the proper testing and application of the NN approximations of a target mapping (2.1). Figure 2.1 illustrates the mapping (2.1), its domain, D , and range, R .

2.1.2 Multilayer Perceptron NNs

The simplest MLP NN with one hidden layer is a generic analytical nonlinear approximation or model for the target mapping (2.1). The MLP NN uses for the approximation a family of functions like:

$$y_q = NN(X, a, b) = a_{q0} + \sum_{j=1}^k a_{qj} \cdot t_j; \quad q = 1, 2, \dots, m \quad (2.2)$$

where

$$t_j = \phi(b_{j0} + \sum_{i=1}^n b_{ji} \cdot x_i) \quad (2.3)$$

and x_i and y_q are components of the input and output vectors respectively, a and b are fitting parameters or NN weights, and ϕ is a so called activation or “squashing” function (a nonlinear function, often specified as the hyperbolic tangent), n and m are the numbers of inputs and outputs respectively, and k is the number of the nonlinear basis function, t_j (2.3), in the expansion (2.2). The expansion (2.2) is a linear expansion (a linear combination of the basis function t_j (2.3)) and the coefficients a_{qj} ($q = 1, \dots, m$ and $j = 1, \dots, k$) are the linear coefficients of this expansion. It is essential (see Sect. 2.4.1) that the basis functions t_j (2.3) are nonlinear with respect to inputs x_i ($i=1, \dots, n$) and to the fitting parameters or coefficients b_{ji} ($j = 1, \dots, k$). As a result of the nonlinear dependence of the basis functions on multiple fitting parameters b_{ji} , the basis $\{t_j\}_{j=1, \dots, k}$ becomes a very flexible set of non-orthogonal basis functions that have great potential to adjust to the functional complexity of the mapping (2.1) to be approximated. It has been shown by many authors in different contexts that the family of functions (2.2, 2.3) (i.e., the simplest MLP with one hidden layer) can approximate any continuous or almost continuous (with a finite number of finite discontinuities, like step functions) mapping (2.1) (Cybenko 1989, Funahashi 1989, Hornik 1991). The accuracy of the NN approximation or the ability of the NN to resolve details of the target mapping (2.1) is proportional to the number of basis functions, k (Attali and Pagès 1997). Taking into account these results, in the NN-TVS, we implemented a MLP with one hidden layer represented by (2.2,2.3).

The MLP NN (2.2,2.3) itself is a particular type of mapping (2.1). In the case of the MLP NN, the *computational and functional complexities* (Krasnopolsky 2013) of the NN mapping are closely related (they are especially close for NN emulations, see Sect. 2.5), and can be

characterized by the number of fitting parameters a and b in (2.2,2.3). This number, the complexity of the MLP NN, is given by

$$N_c = k \cdot (n + m + 1) + m \quad (2.4)$$

For a set of NNs approximating a particular target mapping (2.1) with a given number of inputs n and outputs m , a good measure of the NN complexity is the number of basis functions, k , that are used. The NN complexity grows linearly with the growth of the dimensionalities of the input space (the number of inputs), n , and the output space (the number of outputs), m . It is noteworthy that, if for each mapping output y_q we construct a polynomial approximation, such a multidimensional polynomial of order P would have n^P unknown fitting parameters (Bishop 2006). Therefore, in the case of polynomial approximation, the computational complexity of the approximation for the entire mapping (2.1) is $m \cdot n^P$, which is a power law growth. The power law growth is slower than an exponential growth; but it is very fast and leads to the curse of dimensionality. Thus, the polynomial approximation is of limited practical utility for multidimensional function and mapping approximations. NNs manage to address the curse of dimensionality and, due to the aforementioned linear dependence on the dimensionality of the input space, remain a practical approximation (or model) even for high dimensional mappings.

2.1.3 NNs in their traditional terms

A pictographic language reminiscent of a data flow chart is used traditionally in the NN field starting with the founding work by McCulloch and Pitts (1943). In this work devoted to the mathematical modeling of a *neuron*, a single cell in a neural network, a basis function t_j (2.3), was represented as shown in Figure 2.3 (left). A step function was used in this work as the activation function, ϕ .

Then after Rumelhart et al. (1986) introduced the MLP NN, a pictographic representation of the entire NN was introduced (see Fig. 2.2). The neurons are situated into *layers* inside the MLP NN. The input layer is in a sense a symbolic layer. Input neurons do not perform any numerical function; they simply distribute inputs to neurons in the following hidden layer. The hidden layer (there can be several) is usually composed of nonlinear neurons (eq. (2.3) and Fig. 2.3 left). The neurons in the output layer are usually linear ((eq. (2.2) and Fig. 2.3 to the right). The

connections (arrows) in Figure 2.2 correspond to the NN weights, the name used for the fitting parameters a and b in NN jargon.

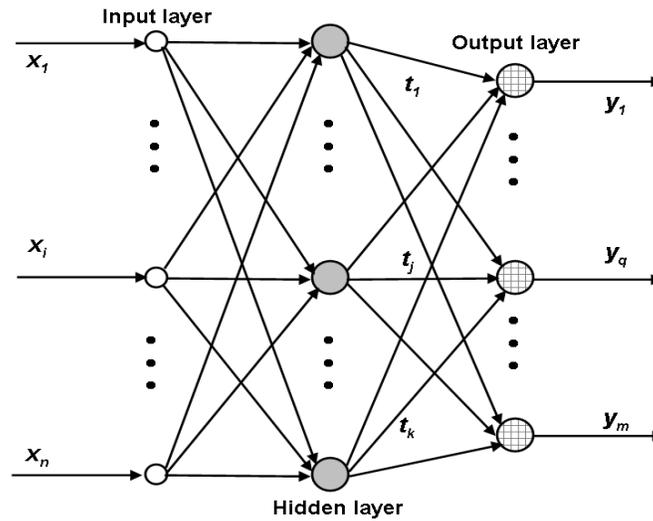


Figure 2.2. The simplest MLP NN with one hidden layer and linear neurons in the output layer.

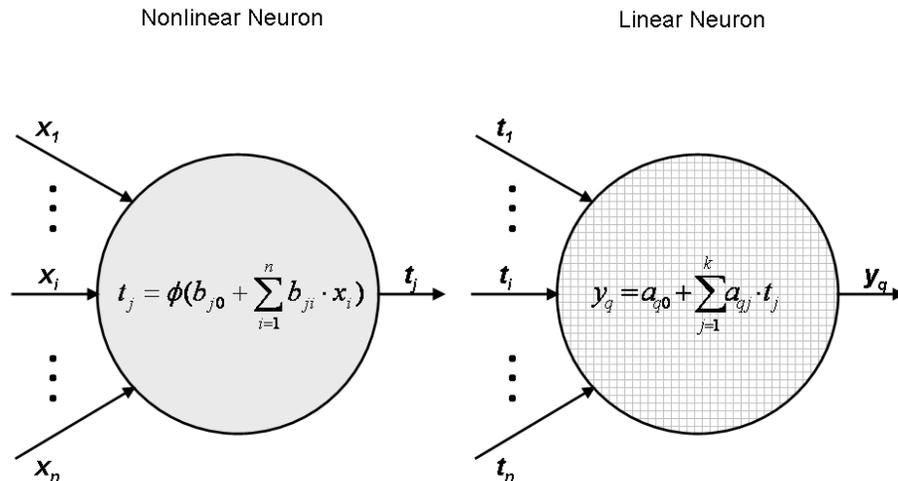


Figure 2.3. The right figure shows linear (eq. (2.2)) and the left – nonlinear (eq. (2.3)) neurons.

As was explained in the previous section, in the NN-TVS the simplest type of MLP NN that has one hidden layer and the output layer with linear neurons has been implemented. Such

architecture is sufficient for the approximation of any continuous (or almost continuous) mapping (Cybenko 1989). More than one hidden layer and nonlinear neurons in the output layer may be introduced to solve specific problems (e.g., (Hsieh 2009)) or for practical convenience. For the simplest MLP NN considered here, there is a one-to-one correspondence between eqs. (2.2,2.3) and Figs. 2.2 and 2.3.

2.2 Preparation for NN Training

2.2.1 Training set

In a practical application, a target mapping (2.1), is usually represented and presented to the NN by a data or training set that consists of N pairs of input and output vectors X and Y ,

$$C_T = \{X_p, Y_p\}_{p=1, \dots, N} \quad (2.5)$$

where $Y_p = M(X_p) + \zeta_p$, $X_p \in D$ and $Y_p \in R$, ζ_p represents any errors associated with the observations or calculation with a probability density function $\rho(\zeta)$. The set C_T can also be considered as a combination of two rectangular matrixes,

$$C_T = \{C_X, C_Y\}, \quad (2.5a)$$

where C_X is a matrix of dimensionality $N \times n$ composed of all input vectors X , and C_Y is a matrix of dimensionality $N \times m$ composed of all output vectors Y . The format of the training set that is used in the NN-TVS is described in Sect. 3.3. The training set is all that the NN “knows” about the target mapping that it is expected to approximate. This is the reason why NN belongs to a class of *data driven* or *learning from data* methods (Cherkassky and Mulier 2007).

The training set represents the mapping (2.1) for the NN and, therefore, it has to be *representative*. It means that the training set has to have a sufficient complexity to represent the complexity of the target mapping, allowing the NN to achieve the desired accuracy of the approximation of the target mapping. The set should have a sufficient sample size, N , of properly distributed data points that adequately resolve the functional complexity of the target mapping (2.1). In a perfect world, the training set should have finer resolution where the target mapping is not smooth and coarser resolution where it is smoother, namely, the domain D should be properly sampled. Certainly, it may be over-sampled but not under-sampled. The

fundamental question remains, however, as to just how we should measure this target mapping smoothness (or complexity) in order to obtain the desired results (DeVore 1998). The interrelations and correlations between inputs, as discussed earlier, simplify the sampling task for cases of high input dimensionality, reducing the size and the effective dimensionality of the domain.

The representativeness of the training set is a necessary condition for a good NN generalization (interpolation), for obtaining acceptable performance of a NN. Kon and Plaskota (2001) introduced a qualitative measure for the representativeness or necessary complexity of the training set. They introduced what they called *informational complexity*, which specifies the number of observations necessary and sufficient to construct a NN approximation under accepted assumptions. In an ideal situation *there should be a correspondence between the functional complexity of the target mapping (2.1), the complexity (2.4) of the approximating NN, and the informational complexity (number and distribution of data points) of the training set (2.5)*. Unfortunately, there are no general recipes for most practical applications. The only useful relationship that can be found in the literature is $N > N_c$. Actually, this relationship has a simple statistical interpretation; the number of unknown parameters in the model (the number of weights in NN or the complexity of the NN (2.4)) should not exceed the number of data points, N , in the training set (2.5). It is noteworthy that NNs with $N_c \geq N$ can be considered if regularization procedures are applied during the training.

As mentioned above and is discussed below, the training set should be somewhat redundant; however, an excessive redundancy with almost identical records in the training set does not improve the NN training and does obviously increase the NN training time. Various methods have been proposed to deal with such a redundancy of the training set. For example, Chevallier et al. (2000) introduce a sampling method that uses Euclidian or non-Euclidian distances in the input space \mathfrak{R}^n to eliminate almost identical records from the training set. A dissimilarity index is introduced in this case,

$$D_n(X_i, X_j) = \|X_i - X_j\|$$

where $\|\dots\|$ denotes a norm in the input space \mathcal{R}^n . Only records with the dissimilarity indexes $D_n > d$ (a predefined distance in \mathcal{R}^n) have been included in the training set.

Two distinct types of data (observed and simulated) are usually employed in the atmospheric and oceanic applications. The first type of data is based on observations. These data usually contain a significant level of observational noise ζ . It is noteworthy that for an ill-posed problem (e.g., satellite retrievals) even a small level of noise in the data may lead to significant errors in the NN emulations. In the case of observed data, the sampling of the target mapping domain is controlled by the observation setup, technique, and conditions. Actually, in this case, because the target mapping is represented implicitly by the available observed data, the accuracy of the NN approximation and the ability of the NN approximation to resolve the target mapping are limited and determined by the observation setup, technique, and conditions. Two important problems when working with observed data are the data quality control and missing data handling (see Sect. 2.3.2 for a brief discussion of the missing data problem). There is usually little we can do to improve or expand the data set in the case of observed data except to fuse it with simulated (model produced) data if such data can be produced.

If an explicit theoretical (based on first principles) or empirical model for the target mapping (2.1) is available, it can be used to simulate the data set (2.5). With simulated data, we have significantly more control over the sampling of the target mapping domain (the number and distribution of the data points) and, as a result, on the NN accuracy and the ability of the emulating NN to resolve the target mapping. The level of noise in the simulated data is usually lower than that in the observed data. The simulated data do not have missing data per se; however, in some application a problem may arise, which is similar to the missing data problem (see Sect. 2.3.2). The simulated and observed data can, in principle, be merged or fused together to form an integrated data set using an appropriate technique that is able to account for the different error statistics and statistical properties of these two data types. One example of properly fused data is the analyzed data produced by a Data Assimilation System.

2.2.2 Selection of the NN architecture

To approximate a particular target mapping (2.1) with the MLP NN (2.2,2.3), we should first select the NN architecture or topology, the number of the inputs n , the outputs m , and the number

of neurons k in the hidden layer. For each particular problem, n and m are determined by the input and output dimensionalities of the target mapping (the dimensions of the input and output vectors X and Y). Here we treat an entire mapping (2.1) as an elementary/single object and approximate its functionality (an input-output relationship) in its entirety. Practical implementation of this approach allows for multiple solutions in terms of the number of NN designs that can be used for an approximation. As a result, the MLP NN expressed in Eqs. (2.2,2.3) can be implemented as a single multi-output NN with m outputs, m single-output NNs, or several multiple-output NNs with the total number of outputs equal to m .

Approximating the target mapping with a single NN is a convenient solution because of the simplicity of its design. It also has a great advantage in terms of speeding up the calculations when the outputs of the mapping and, therefore, the outputs of the approximating NN are significantly correlated. In the case of a single NN (2.2,2.3) with many outputs, all the outputs are different linear combinations of the same basis functions t_j , or hidden neurons. Fewer neurons are required to approximate a particular number of correlated outputs than to approximate the same number of uncorrelated outputs. Thus, in the case of correlated outputs, one NN per approximation solution has a lower complexity N_c (2.4), and provides significantly higher performance at the same approximation accuracy than a battery (an array) of m single-output NNs (Krasnopolsky and Fox-Rabinovitz 2006, Krasnopolsky et al. 2014). Also, a single emulating NN solution is less complicated in terms of NN training requirements. It leads to a lower dimensionality of the training space (see Krasnopolsky et al. (2014) for examples and discussion).

The number of hidden neurons k , that determines the complexity (2.4) of the approximating NN in each particular case should be determined when taking into account the complexity of the target mapping to be approximated. The more complicated and nonlinear the mapping, the more hidden neurons k are required (Attali and Pagès 1997) (or the higher the required complexity N_c and nonlinearity of the NN) to approximate this mapping with the desired accuracy or resolution. There is always a trade-off between the desired approximation accuracy of the target mapping and the complexity of the NN emulation. Though, from our experience, the complexity k of the approximating NN should be carefully controlled and kept to the minimum level sufficient for the desired accuracy of the approximation to avoid overfitting and to allow for a smooth and

accurate interpolation (see the discussion in Sect. 2.4.3). Unfortunately, in this regard, there are no universal rules or recommendations that can be given. Usually k is determined based on experience and experimentations. In the NN-TVS the NN architecture is specified in the script *startr* (see Sect. 3.2). For problems that have been solved using the NN-TVS k varied from 3 to 300 depending on the problem (Krasnopolsky 2013).

The possibility of choosing among many topological solutions, from a single NN with m outputs to m single-output NNs, demonstrate the internal flexibility of the NN technique. This additional flexibility can be effectively used in many applications (e.g., to create ensembles of NNs).

2.2.3 Normalization of the NN inputs and outputs

Another degree of flexibility is provided by the availability of different *normalizations* for NN inputs and outputs. NN inputs are usually normalized to an interval $[-a,a]$, using a simple equation,

$$\tilde{x}_i = a_i \cdot \frac{(2 \cdot x_i - x_i^{\max} - x_i^{\min})}{(x_i^{\max} - x_i^{\min})} \quad (2.6)$$

where x_i is the i -th NN input before and \tilde{x}_i after the normalization. If all a_i are equal to 1, then all inputs are normalized to the interval $[-1,1]$. By selecting different a_i for different inputs, we can change the sensitivity of NN to the variability of a particular input. In the NN-TVS the normalization of inputs and outputs is performed using the IDL routine YSCALE (see Sect. 3.4).

For NN with a single linear output, normalization of the output is relatively straightforward. Any traditional normalization, like normalizing over the interval $[-a,a]$ via (2.6) or using the following normalization,

$$y' = \frac{y - \bar{y}}{\sigma} \quad (2.7)$$

where \bar{y} is the mean value of y and σ is its standard deviation (SD), can be used and leads to similar approximation errors.

For a single NN with multiple outputs, the normalization of the outputs affects the approximation accuracy and NN performance more significantly than in the case of a single output NN. Normalization similar to (2.7) for the case of multiple outputs can be written as,

$$y_q' = \alpha \cdot \frac{y_q - \bar{y}_q}{\sigma_q} \quad (2.8)$$

where \bar{y}_q and σ_q are the mean and SD of the q^{th} output, y_q , and $\alpha \leq 1$ is introduced to accelerate the training of the linear weights in the output layer of the NN. This normalization improves approximation accuracies for small outputs; however, if these outputs are noisy, it propagates the noise to other outputs. Normalization (2.8) also distorts nonlinear relationships and reduces nonlinear correlations that may exist between outputs.

An alternative normalization,

$$y_q' = \alpha \cdot \frac{y_q - \bar{y}}{\sigma} \quad (2.9)$$

where σ is the SD and \bar{y} the mean value for all outputs, can be also employed in the case of NNs with several outputs. This normalization preserves correlations between outputs. Via taking into account these correlations, the normalization allows to reduce complexity and to improve performance of the emulating NN.

In the case of multiple outputs, the four different normalizations (2.6 – 2.9) lead to very different approximation errors and to different types of error distributions between outputs (see Krasnopolsky (2013) for examples). For different NN applications, certain types of error distributions may be desirable; for example, smaller absolute or relative errors may be preferable. Different output normalizations in the case of a single emulating NN with multiple outputs may provide an additional tool for obtaining the desired result. This topic is also discussed by Krasnopolsky (2013). The IDL program YSCALE performs an input normalization (2.6) and various output normalizations.

2.2.4 Constant inputs and outputs

In practical applications some components of input and/or output vectors X and Y of the mapping (2.1) have constant or almost constant values (their variability is very small). In such cases, when the mapping is emulated with NN, constants should not be included in inputs or outputs because (1) they carry no information about input/output functional dependence and (2) if not removed they introduce additional noise in training and result in additional approximation errors. As for values that are almost constant, these small signals may be in some cases not a noise but very important signals; however, in such cases they should be specially treated to be used as NN inputs or outputs. For example such inputs or outputs should be converted to their anomalies (i.e., their means should be subtracted); then the anomalies should be used as inputs or outputs for the emulating NN. On the other hand, depending on the level of uncertainty in the problem, information that these small signals may provide may be well below the level of uncertainty, and may be in many cases practically useless.

2.3 NN training

After the NN architecture (topological parameters n , k , and m) is defined, the weights (a and b) can be found using the training set C_T (2.5) and the maximum likelihood method (Vapnik 1995) by maximizing the likelihood functional

$$L(a, b) = \sum_{i=1}^N \ln \rho(\xi_i) \quad (2.10)$$

with respect to the free parameters (i.e., the NN weights) a and b . Here, $\rho(\xi)$ is the probability density function for the approximation errors $\xi_i = Y_i - NN(X_i, a, b)$ and the summation is performed over the N records in the training set. If the errors ξ_i are normally distributed, maximizing the likelihood functional (2.10) leads to the minimization of the least-square error function (also called the loss, or risk, or cost function) with respect to the NN weights given by $W(a$ and $b)$,

$$E(W) = \frac{1}{N} \sum_{i=1}^N \xi_i^2(W) = \frac{1}{N} \sum_{i=1}^N (Y_i - Z_i)^2 \quad (2.11)$$

where $Z_i = NN(X_i, W)$, E is the total error calculated over the entire training set (all N records of the training set are included) and $\xi_i^2 = (Y_i - Z_i)^2$ is a square error corresponding to the i^{th} record in the training set. The procedure of minimization of the error function (2.11) is usually called *NN training*. Minimizing the error function is performed in the W -space (the space of NN weights or the training space), which has a dimensionality equal to the number of NN weights, N_C (2.4). It is noteworthy that for a probability density function $\rho(\xi)$ other than Gaussian, the error function should be derived from the maximum likelihood functional (2.10). The error function may be significantly different than the least-square error or loss function (2.11) (Liano 1996). But in the majority of applications the least-square error function (2.11) is applied because it is simple and analytically differentiable.

Optimal values for the weights are obtained by minimizing the error function (2.11); this task is a nonlinear minimization problem. A number of methods have been developed for solving this problem (Bishop 1995, Haykin 2008). Here we briefly outline one of them, a simplified version of the steepest (or gradient) descent method known as the back-propagation training algorithm.

The back-propagation training algorithm is based on the simple idea that searching for a minimum of the error function (2.11) can be performed step by step iteratively and that at each step we should increment or decrement the weights in such a way as to decrease the error function. This can be done, for example, using the following simple steepest descent rule,

$$W^{(n+1)} = W^{(n)} - \eta \frac{\partial E(W^{(n)})}{\partial W} \quad (2.12)$$

where W is either one of two weights (a or b), $W^{(n+1)}$ is an adjusted or updated weight, $\eta > 0$ is a so-called learning constant, and $W^{(n)}$ is the weight at the previous n^{th} iteration. The total error function E or a one record error function ξ_i^2 can be used in (2.12) which leads to different training procedures, *batch* training or *sequential* (or *on-line*) training (see next Sect. 2.3.1).

Using (2.11), (2.2,2.3), and the chain rule of differentiation, the derivative in (2.12) can be expressed analytically through the derivative of the activation function ϕ , and through the weight values at the previous iteration step (Haykin 1994, Bishop 1995, 2006). At the first step when we

do not have weights from a previous training iteration, a weight initialization problem arises that is familiar to those who use various kinds of iterative schemes. Many studies have been devoted to weight initialization (e.g., Nguyen and Widrow 1990, Wessels and Bernard 1992). Most of these procedures initialize the NN weights with small random numbers. Wessels and Bernard (1992) for example, generate random weights in the range $[-3/\sqrt{n}; 3/\sqrt{n}]$, where n is the number of NN inputs. In the NN-TVS, the Nguyen and Widrow initialization is performed by an IDL program NWGINI and the Wessels and Bernard initialization – by an IDL WGTINTI program.

The nonlinear error function (2.11) has multiple local minima. Moreover, due to the symmetry of MLP NN in the weight space, for each of these local minimum there exist $k! \cdot 2^k$ ($k!$ is k -factorial) clone local minima with exactly the same error (Chen et al. 1993). The back-propagation algorithm converges to a local minimum, as does almost any algorithm available for solving the nonlinear optimization problem (NN training). Usually, multiple initializations (even multiple initialization procedures) are applied to avoid shallow local minima and to choose a local minimum with a sufficiently small error.

2.3.1 Batch training and sequential training

If in eq. (2.12) the total error function E is used to calculate the new adjusted weights; therefore, the entire training set has to be processed at each training iteration. Thus, at each training step, the weights are shifted in the direction of the greatest decrease in the total error function. This type of training that uses the entire data set to calculate each weight adjustment is called *batch* training.

Alternatively, we can use error function ξ_i^2 in eq. (2.12), which is one record (i.e., one pattern) error, for the weight adjustment; that is, the weights are updated after each data record from the training set is presented to the NN. This training approach is called *sequential* or *on-line* training. Sequential training works with the training set record by record until all patterns have been presented once to NN, which is called an *epoch*, then the process may be repeated many times (many epochs) cycling through the records of the training set in sequence or selecting patterns randomly. Thus, in the batch training weights are updated once per epoch, while in the sequential training weights are updated after each data pattern (record) or N times per epoch.

The batch approach is meaningful from the statistical perspective. It can be effectively parallelized for executing on supercomputers. It is often effective, and certain second-order optimization algorithms work better with batch training (Hsieh 2009); however, certain problems related to this algorithm can arise; they are discussed by Bishop (2006) and Hsieh (2009). The sequential approach has the obvious advantage in the case of long training sets: it works with one record at a time, which makes it independent of the number of N patterns in the training set. Thus, the sequential approach can be successfully used when dealing with data that arrive in real time. Each new data record can be used independently to update the NN weights on-line. Also sequential training allows avoiding local minima of the total error function. A version of the sequential training is currently implemented in the NN-TVS FORTRAN code *trainl*.

2.3.2 Missed inputs and outputs

In practical applications some elements of the data matrixes C_X and/or C_Y , constituting the training set C_T (2.5a), may be missed or corrupted. Obviously, this problem often occurs when working with observed data. Though, the problem of missing data or a similar problem also arises when using simulated data. In one of the NN applications considered by Krasnopolsky et al. (2009), the mapping (2.1) describes the entire atmospheric physics; the output vector in this case includes, among others, two physical parameters: the land temperature at a given point and the ocean temperature at the same point. Obviously, at a particular location, only one of these two parameters is valid, and another one is missed. Thus, in this case, each record in the training set simulated by a numerical model contains at least one missing value.

Missing data is an important technical problem because the majority of multivariate data modeling and analysis techniques (including NNs) require complete data sets (all variables have to be represented for each data record). Rates of missing data less than 1% are usually considered trivial, 1 to 5% - manageable, a rate of 5 to 15% requires sophisticated methods to handle it, and more than 15% may have a severe impact on the quality of the model (Luengo et al. 2010).

Several approaches have been proposed to treat missing data. The most economical way of dealing with the problem and to obtain complete data set is by deleting the records with missing data, i.e., working with a subset of the training set. This method, however, may become

unacceptable in the case of small data sets, especially in the case of high dimensional input and output vectors X and Y . Indeed, if only one element (one variable) of either vector is missing, the entire record (X_p, Y_p) , which contains $n + m - 1$ acceptable variables, has to be removed. In the previous example that involves atmospheric physics, application of this method will lead to deleting the entire data set. Fortunately, a number of less economical but more sophisticated methods have been developed to deal with missing data (Richman et al. 2009). For example, a more sophisticated approach would be to use the maximum likelihood procedure, where the parameters of a model for the complete data are estimated. This model can be used to impute missing data. Finally, in the majority of cases, data set components are not independent from each other. Thus, through the identification of relationships between components, missing values can be determined and imputed.

The detailed discussion of the problem of missing data goes beyond the scope of this Note. It is discussed by Richman et al. (2009) and Luengo et al. (2010); various methods of data imputation or replacement of missing data are considered in these works and in papers cited there. In this Note, we mention one simple method of dealing with missing output data that is used in (Krasnopolsky et al. 2009). This method is equally effective if one or more components of the output vector Y_i are missing. In the application of this method the error function (2.11) is modified by introducing in the error function a binary matrix α_{iq} ,

$$\bar{E}(W) = \frac{1}{N} \sum_{i=1}^N \sum_{q=1}^m \alpha_{iq} \cdot (y_q^i - z_q^i(X_i, W))^2 \quad (2.11a)$$

where matrix, α , is defined in accordance with the following rule: for the training record number i ,

$$\alpha_{iq} = \begin{cases} 1, & \text{if the output number } q \text{ is defined} \\ 0, & \text{if the output number } q \text{ is missing} \end{cases}$$

Thus, the method avoids deleting the entire record number i from the training set. All components of the vector that are not missing can be used for training. Missing components are simply not included in the error function (they are included with zero weight). The weight matrix α_{ij} is implemented in the NN-TVS.

2.3.3 Overfitting and regularization

In this section we discuss the problem of overfitting for two different cases. First, when the level of noise in the training data is low and a NN or any other nonlinear statistical model approximates the data well, it may occur that between the data points and/or at the ridges of the domain the NN exhibits an unpredictable behavior (e.g., wild oscillations). Second, when dealing with noisy data containing outliers, the model, if the training is excessive, may fit the noise and outliers as well as the desired data. There are a number of reasons for overfitting. For example, the NN complexity, N_C , can be unreasonably high and approach or exceed the number of data points, N , in the training set. Also, the training set can be long (large N) but redundant; a simple target mapping can then be oversampled. In this case, NNs of high complexity can be selected based on the large value of N ; however, because of the redundancy in the data, overfitting will occur.

One of the classic manifestations of overfitting is the convergence of the training process to a local minimum with large weights of alternating signs. To avoid converging to such local minima, *regularization* is often applied (Haykin 1994, Bishop 1995, 2006), which involves adding a penalty term to the error function (2.11). A modified (or regularized) error function can be written as,

$$\tilde{E}(W) = E(W) + \lambda \cdot \sum_j W_j^2 \quad (2.11b)$$

The first term in eq. (2.11b) is the standard error function (2.11), the second term is the regularization term, and λ is a coefficient that reflects the relative importance (or strength) of the regularization term. It is obvious that the regularization term in eq. (2.11b) penalizes large weights providing guidance for the training procedure, based on the regularized error function (2.11b), to a solution that corresponds to a local minimum with smaller weights and thus preventing overfitting due to large weights.

It is noteworthy that in the case of complex NNs with a large number of weights, the dimensionality of the training space, N_C , is very high (hundreds of thousands in some applications discussed in Krasnopolsky 2013). As a result, the number of local minima in the error function can be very large. In such situations, NN training usually leads to one of the

closest (to the initialization point) local minima. If initialization procedure is used that initializes NN weights with small random numbers (e.g., Nguyen and Widrow 1990), the training process will be attracted to a local minimum, which is close enough to the initialization point, and where the weights will still be relatively small. Thus, application of an initialization procedure, like one just described, may make the use of regularization (2.11b) unnecessary for complex NNs.

2.4 Advantages and Limitations of the NN Technique

Here we summarize the advantages and limitations of the MLP NN approach as applied to the emulation of complex multidimensional mappings (2.1). It is noteworthy that the majority of limitations we discuss here are not limitations of the MLP NN technique *per se*. These limitations are inherent with regard to nonlinear models, nonlinear approximation techniques, and nonlinear statistical approaches in general (Cheng and Titterington 1994). Also, the same feature of the NN technique that gives this technique a significant advantage under the normal circumstances is sometimes responsible for some of the limitations on the NN technique under special conditions. We will proceed with the discussion while keeping these two points in mind.

2.4.1 Flexibility of the MLP NN

The MLP NN is a universal and very flexible approximator. The great flexibility of the MLP NN is due to the fact that the basis functions (hidden layer neurons) t_j (2.3) are adjustable. They contain many internal nonlinear parameters b that can be adjusted during training process. Thus, the basis functions are not specified a priori; they are determined during the training process and “optimized” for a particular mapping to be approximated. Barron (1993) showed that, for certain classes of mappings, a linear combination of such adjustable basis functions can provide an accurate approximation with far fewer functions than a linear combination of any fixed or rigid basis functions that contain no adjustable nonlinear parameters. Similar results were obtained by Krasnopolsky and Kukulín (1977). This is one way for the MLP NN to escape the curse of dimensionality.

Another way to look at the importance of the adjustable basis t_j , is to demonstrate the independence of the approximation error with respect to the dimensionality n of the input space.

When flexible basis functions (2.3) are used, the approximation error, $E \leq \frac{\alpha}{k^p}$, where $\alpha > 0$, $p > 0$ and p is independent of n . In contrast, for approximations using fixed basis functions, the approximation error is $E \leq \frac{\alpha}{k^n}$ for the same class of mapping (Barron 1993, Cheng and Titterington 1994). Therefore, for a fixed basis expansion, when the number of inputs increases, one needs more and more basis functions (the number of basis functions, k , has to increase) to achieve the same accuracy of approximation. Thus, for the MLP NN, it is the number of hidden neurons k , not the dimensionality of the input space that determines the accuracy of the approximation.

The flexibility of the MLP NN may also lead to undesired consequences. The basis functions t_j are very flexible, non-orthogonal, and overlapping. These factors may lead to non-optimality or redundancy in the NN architecture. As a result, some of the hidden neurons may contribute very little to the approximation and could be removed by “pruning” without a significant impact on the approximation accuracy. Pruning and similar techniques (Bishop 1995, Haykin 1994) have been developed to optimize the NN architecture and complexity.

Thus, the flexibility of the MLP NN technique, if not properly implemented and controlled, may lead to unwanted consequences like overfitting (fitting the noise in the data), unstable interpolation, and uncertain derivatives.

2.4.2 NN training, nonlinear optimization, and correlation of inputs and outputs

NN training, as described in Sect. 2.3, is an iterative procedure that does not involve any matrix inversion. It is robust, insensitive to correlations (multi-collinearities) in input and output data, and always leads to a solution for the NN weights. On the other hand, as a nonlinear optimization, NN training always has multiple solutions that correspond to multiple local minima in the error or loss function (2.11). Multi-collinearities in the input and output data lead to an equalization of local minima, especially in the case of a higher input dimensionality. Therefore, multi-collinearities in input and output data partly alleviate the problem of seeking the local minimum with the smallest error among multiple local minima. From the point of view of the approximation problem, all of these local minima give similarly good solutions because the

approximation errors for these minima are small and approximately the same. On the other hand, these local minima, which are almost equivalent in terms of the approximation error, give different solutions in terms of the NN weights. These different NNs may lead to different interpolations and different derivatives. Thus, because of the equalization of errors corresponding to different local minima, the approximation error may not be a sufficient criterion; hence, using additional criteria may be necessary for selecting solutions with acceptable interpolation properties and derivatives.

2.4.3 NN generalization: interpolation and extrapolation

One of the vaguest terms in NN parlance is “generalization” or “generalization ability”. This term came from the field of cognitive science and implies an acceptable performance of the trained NN for new values of inputs that were not included in the training set. Yet, it is clear that there are at least two different cases of generalization. In the first case, new inputs are located inside the domain, D , “between” the training data points. In the second case, new inputs are located beyond the area covered by the training set, namely close to, or outside of the boundary of the domain D . The first case corresponds to interpolation; and the second, to extrapolation.

It is well known that nonlinear extrapolation is an ill-posed problem and its solution may require regularization (introducing additional information) (Vapnik 1995). We will not discuss nonlinear extrapolation here. However, even smooth interpolation is not guaranteed if the only criterion used for NN training is small approximation error (2.11). Moreover, multiple local minima with similarly small approximation errors may still lead to different interpolations. If the NN complexity is not controlled, overfitting may occur that may lead to poor interpolations, e.g., significant oscillations between training data points. As mentioned in Sect. 2.2.1, the representativeness of the training set is a necessary condition for acceptable interpolations. Additional measures for improving the interpolation ability of the NN approximation are discussed in Sect. 2.5.

2.4.4 NN Jacobian

The NN Jacobian, J , is an $m \times n$ matrix of the first derivatives of the NN outputs over the inputs,

$$J = \left[\frac{\partial y_q}{\partial x_i} \right]_{\substack{q=1,\dots,m \\ i=1,\dots,n}} \quad (2.13)$$

may be useful in many cases. For example, in data assimilation applications the Jacobian is used to create an adjoint (a tangent-linear approximation) of the target mapping. The Jacobian is also instrumental in statistical sensitivity, robustness, and error propagation analyses of the target mapping and its NN emulation. An inexpensive, simple computation of the NN Jacobian by analytical differentiation of (2.2,2.3) is one of the advantages of the NN approach. It is important to understand that the Jacobian is not trained; it is simply calculated through a direct differentiation of a trained NN. In this case the statistical inference of a Jacobian represents an ill-posed problem, and it is not guaranteed that the derivatives will be sufficiently accurate. Moreover, the existence of multiple minima of the error function with similar approximation errors and different NN weights implies that there exist multiple solutions for emulating NNs that have similar approximation and interpolation errors but different Jacobians.

As mentioned in Sect. 2.4.3, if additional care is taken during the training process, NN emulations can demonstrate acceptable interpolation properties (Krasnopolsky 2013). Thus, on average, the derivatives of these emulations are sufficiently accurate to provide satisfactory interpolations; however, for certain applications, such accuracy of a NN Jacobian may be not sufficient. For those applications that require an explicit calculation of the NN Jacobian, several solutions have been offered and investigated:

1. The Jacobian (or the entire adjoint) can be trained as a separate NN (Krasnopolsky et al. 2002). Generation of a data set for training a Jacobian or adjoint is usually not a significant problem in those cases where simulated data are available.
2. An ensemble approach can be applied that uses an ensemble of NN emulations with the same architecture corresponding to different local minima of the error function, or uses an ensemble of NN emulations with different numbers of hidden neurons (different complexities) to stabilize the NN Jacobian or to reduce the uncertainties of the NN Jacobian (Krasnopolsky 2007).

3. The mean Jacobian can be calculated over the entire data set (Chevallier and Mahfouf 2001) and used for further calculations, if necessary.
4. Regularization techniques like “weight smoothing” (Aires et al. 1999) or principal component decomposition (Aries et al. 2004b) can also be used to stabilize the Jacobians.
5. The Jacobian can be included in the NN architecture as additional outputs that can be trained
6. The error or cost function $E(W)$ (2.11), which is minimized in the process of NN training can be modified to accommodate the Jacobian; in other words, the Euclidian norm, which is usually used for calculating the error function, should be changed to the first order Sobolev’s norm error function, according to

$$E_J(W) = E(W) + \lambda \cdot \sum_{i=1}^N \|J(X_i) - J_{NN}(X_i, W)\|^2$$

where J is the Jacobian matrix of mapping (2.1) (observed or simulated), J_{NN} is the Jacobian matrix of emulating NN, $\|\dots\|$ denotes the matrix norm, and λ is a constant reflecting relative importance of the two terms constituting the modified error function $E_J(W)$. With this change from Euclidian to Sobolev’s norm, the NN is trained to approximate not only the target mapping (as with the Euclidian norm) but also the mapping’s first derivatives. This solution does not change the number of the NN outputs; however, it may require using more hidden neurons and may significantly complicate the minimization during the training since the complexity of the error function increases. Hornik et al. (1990) have shown that the function of Sobolev’s space with all their derivatives can be approximated by a NN. This and other similar theoretical results are very important because they prove the existence of the approximation, however, they do not suggest explicit approaches. Some explicit approaches have been presented elsewhere (Cardaliaguet and Euvrard 1992, Lee and Oh 1997).

Solutions 5 and 6 require an extended training set that includes first derivatives. This requirement cannot usually be met when working with high dimensional mappings represented by observed data. When working with data simulated by a physically-based model, it is usually

not difficult to additionally simulate the derivatives. Finally, it should be mentioned that Jacobian modeling for large NNs still remains an open issue. Currently in the NN-TVS the Jacobian is not trained.

2.4.5 Multiple NN emulations for the same target mapping and NN ensemble approaches

Nonlinear models and approximations have many nonlinear parameters that could change during the process of generating solutions (fitting the data), which makes the models very flexible and easily adjustable to a selected target mapping. Different combinations of these parameters may lead to multiple solutions with the same or almost the same values of approximation errors. The existence of multiple solutions is a generic property of nonlinear models. The multiple solutions may be almost identical or similar in terms of a particular criterion (e.g., error function) that is used to obtaining the solutions. At the same time these models (e.g., NNs) may be different in terms of other criteria that provide complementary information about the target mapping. The availability of multiple solutions may at times be inconvenient and lead to uncertainties, e.g., the necessity of introducing an additional step to use additional criteria to select a single “optimal” model (solution). On the other hand, the availability of multiple models (e.g., NN emulations), providing complementary information about the target mapping, opens up opportunities to use an ensemble approach. It allows integration of the complementary information contained in the ensemble members into an ensemble that collectively “knows” more about the target mapping than does any of the individual ensemble members (individual NN emulations). Thus, an ensemble of learning models consisting of many members is capable of providing a better description of the system than any individual member.

Since the early 1990's, many different algorithms based on similar ideas have been developed for NN ensembles (Hansen and Salamon 1990, Sharkey 1996, Naftaly et al. 1997, Opitz and Maclin 1999, Hsieh 2001). An ensemble of NNs consists of a set of members, i.e., individually trained NNs. They are combined when applied to new data to improve the generalization (interpolation) ability because the previous research has shown that an ensemble is often more accurate than any single ensemble member. Various ways of combining NN ensemble members into the ensemble have been developed (Naftaly et al. 1997). Previous research also suggests that any mechanism that causes some randomness in the formation of the NN members can be used to form a more

accurate NN ensemble (Opitz and Maclin 1999). For example, ensemble members can be created by training different members on different subsets from the training set (Opitz and Maclin 1999), by training different members on different sub-domains of the training domain, by training different members using NNs with different architectures (different numbers of hidden neurons) (Hashem 1997), or by training different members using NNs with the same architecture but different initial conditions for the NN weights (Maclin and Shavlik 1995, Hsieh 2001).

In the approximation of a complex mapping (2.1), the members of the ensemble are separately trained approximating NNs, which provide different accuracies of approximation for the target mapping and different interpolations. Thus, we can expect that the ensemble average will provide higher quality approximations and interpolations than the individual members.

2.4.6 Estimates of NN parameters uncertainty

The NN technique is a nonlinear statistical approach. As with any statistical approach, the NN technique is expected to provide not only an estimate of model parameters and outputs (through the minimization of an error or loss function) but also an estimate of the uncertainties in the NN weights and outputs. Because of the nonlinear nature of NNs, estimation of the NN uncertainties is a more complicated problem than that in the linear case. On the other hand, during the last decade progress has been made in this field for the cases of both the MLP NN with a single output (MacKay 1992, Bishop 1995, Neal 1996, Nabney 2002), and for multiple outputs (Aires et al. 2004a). Various Bayesian methods have been used in these studies for estimating the uncertainties of NN parameters (weights).

2.4.7 NNs vs. physically based models; NN as a “black box”

Outputs of a physically based (PB) model may be related or correlated due to the physical laws (e.g., various conservation laws) and/or equations implemented in the model. NN emulations do not reproduce these relationships exactly. An accurate NN emulation approximates aforementioned relationships and/or correlations with an accuracy limited by the approximation error. If a higher accuracy is desired, a procedure similar to the regularization procedure (2.11b) can be used to include the error in the desired condition as a penalty function in the training process and to minimize this error. Also the desired relationship may be superimposed on NN outputs exactly as a post-processing step (Krasnopolsky 2013).

One of the often mentioned shortcomings of NNs is that they do not offer a straightforward physical interpretation (e.g., Zorita and von Storch 1999), as do PB models or simple linear statistical models, i.e., that NNs are not transparent “black boxes”. In general this is true; it is difficult, if not impossible, to give a physical interpretation to the NN weights, but consider following. NNs are admittedly more complex and less transparent than simple linear models and regressions; however, NNs are never used (or should never be used) for problems that can be solved using linear models. NNs are usually applied to model or emulate multidimensional, complex, nonlinear systems that, in principle, cannot be modeled or emulated by simple linear models. Thus, NNs should not be compared with linear approaches, which are not adequate in these cases. NNs should be compared with other nonparametric multidimensional statistical approaches or complex deterministic numerical models that are used for modeling multidimensional, complex, nonlinear systems. Other nonparametric multidimensional statistical models are also not transparent.

NNs are also not obscure when compared with modern PB models. Modern PB deterministic numerical models have also lost a significant part of their transparency. Here again, when talking about transparency and physical clarity of PB models, one usually keeps in mind first very simplified models developed at the dawn of numerical modeling. Those models evolved to become more adequate to the complexity of systems, which they model. They evolved into modern PB numerical models like General Circulation Models used for weather forecast and climate simulation. Modern models are very complex because they model very complex systems like Earth climate system or its subsystems. They are built of multiple blocks developed by different people and different institutions. The physics implemented in these blocks is parameterized. The parameterized physics is not transparent; it loses direct connections with elementary physical processes, contains many approximations, simplifications, assumptions, and empirical parameters. To make these parameterizations working together coherently in the model, significant number of tuning parameters are introduced. They do not have any physical meaning and are introduced to force different parts of the model to work coherently.

Based on the above, direct comparison of NNs with PB models is clearly a difficult problem. Thus in our view, it is not productive to oppose NNs or other machine learning statistical approaches and first principle models; they should be considered as complementary.

Krasnopolsky (2013) showed that they can be synergistically combined within a hybrid modeling framework, which, if properly implemented, should combine the advantages of both approaches.

2.5 NN Emulations

In this Note, we use the terms *emulating NN*, *NN emulation*, or *NN emulator* for a NN that provides the functional emulation of the target mapping (2.1), including a small approximation error (2.11) for the training set (2.5) and a smooth and accurate interpolation and a limited extrapolation of the training set data inside the domain D . These terms are introduced to distinguish between emulating NNs and approximating NNs. Approximating NNs are usually concerned with an analytic approximation of a data set with a small approximation error (2.11).

When an emulating NN is constructed, in addition to the criterion of a small approximation error (2.11), at least three other criteria presented in Sect. 2.4.3 are used:

1. the NN complexity (2.4) (the number k of hidden neurons) is controlled and restricted to a minimum number that yields a level of accuracy sufficient for a good approximation;
2. independent validation and test data sets are used in the process of training to control overfitting (validation set) and after training to evaluate the interpolation accuracy (test set);
3. a limited and controlled (see Sect. 2.3.3) redundancy is introduced in the training set (additional data points added “in-between” training data points) for improving the NN’s interpolation performance.

The correspondence between the emulating NN complexity (2.4) and target mapping complexity is usually better than that of an approximating NN with the same approximation error. The complexity, N_C , of an emulating NN is usually close to the minimum value possible; thus, the emulating NN is usually faster. Finally, it usually provides a better and smoother interpolation or generalization, better resolution of the target mapping for the same approximation accuracy, and smaller uncertainties in the NN Jacobian.

As it was mentioned in Sec. 2.1.1, NN emulation approximates the target mapping (2.1) in a domain D , which is usually a small subspace of the n -dimensional hypercube limited by minimal

and maximal values of input variables. It means that in the case when we develop a NN emulation of a target mapping given by a PB model, we should keep in mind the following consideration. Both the NN emulation and the PB target mapping will provide outputs for any combination of input parameters inside aforementioned hypercube; but these outputs will be close to each other only inside the domain D , where the NN emulation was trained. Outside this domain the outputs may be very different. Thus, the NN emulation provides an accurate approximation of the target mapping only in the domain of the input space spanned by physically meaningful combinations of inputs.

2.6 Final remarks

In this Section, we discussed the general properties of multidimensional complex mappings (2.1) and MLP NN (2.2,2.3) and also demonstrated relationships between their properties. Both fields are relatively new and so are growing rapidly in terms of the relevant theory and practical applications. In this discussion, we emphasized that a transition from linear statistical tools or models to nonlinear models (like NNs) requires some adjustment in our methodological framework, which may not, at times, be flexible enough to accommodate sophisticated nonlinear approaches.

Many of the advantages of nonlinear statistical techniques may become limitations under certain conditions and in the case of unskillful use. Some of the limitations of nonlinear models may become advantageous when more flexible approaches are employed, by combining different statistical approaches (e.g., NNs and the ensemble approach), and by using additional information. At this time, the NN is probably the only practical statistical learning tool for emulating complex multidimensional mappings.

2.7 References

- Aires F, Prigent C, Rossow WB (2004a) Neural network uncertainty assessment using Bayesian statistics: A remote sensing application. *Neural Comput* 16:2415-2458
- Aires F, Prigent C, Rossow WB (2004b) Neural network uncertainty assessment using Bayesian statistics with application to remote sensing: 3 Network Jacobians. *J Geophys Res.* doi:10.1029/2003JD004175
- Aires F, Schmitt M, Chedin A, Scott N (1999) The “Weight Smoothing” Regularization of MLP for Jacobian Stabilization. *IEEE T Neural Networ* 10: 1502-1510
- Attali J-G, Pagès G (1997) Approximations of Functions by a Multilayer Perceptron: A New Approach. *Neural Networks* 6:1069-1081
- Barron AR (1993) Universal approximation bounds for superpositions of a sigmoidal function. *IEEE T Inform Theory* 39:930-945
- Beale R, Jackson T (1990) *Neural Computing: An Introduction*. Adam Hilger, Bristol, Philadelphia and New York, USA
- Bishop CM (1995) *Neural Networks for Pattern Recognition*. Oxford University Press, Oxford, UK
- Bishop CM (2006) *Pattern Recognition and Machine Learning*. Springer, NY, USA
- Bollivier M, Eifler W, Thiria S (2000) Sea surface temperature forecasts using on-line local learning algorithm in upwelling regions. *Neurocomputing* 30: 59-63
- Cardaliaguet P, Euvrard G (1992) Approximation of a function and its derivatives with a neural network. *Neural Networks* 5, 207-220
- Chen AM, Lu H, Hecht-Nielsen R (1993) On the geometry of feedforward neural network error surface. *Neural Comput* 5:91-927
- Chen T, Chen H (1995a) Approximation Capability to Functions of Several Variables, Nonlinear Functionals and Operators by Radial Basis Function Neural Networks. *Neural Networks* 6:904-910
- Chen T, Chen H (1995b) Universal Approximation to Nonlinear Operators by Neural Networks with Arbitrary Activation Function and Its Application to Dynamical Systems. *Neural Networks* 6:911-917
- Cheng B, Titterton DM (1994) *Neural Networks: A Review from a Statistical Perspective*. *Stat Sci* 9:2-54
- Cherkassky V, Mulier F (2007) *Learning from data*. 2nd ed, John Wiley, Hoboken, NJ
- Chevallier F, Mahfouf J-F (2001) Evaluation of the Jacobians of Infrared Radiation Models for Variational Data Assimilation. *J Appl Meteorol* 40:1445-1461
- Chevallier F, Morcrette J-J, Chérury F, Scott NA (2000) Use of a neural-network-based longwave radiative transfer scheme in the EMCWF atmospheric model. *Q J Roy Meteor Soc* 126:761-776
- Cilliers P (2000) What Can We Learn From a Theory of Complexity? *Emergence* 2: 23-33. doi:10.1207/S15327000EM0201_03

- Cybenko G (1989) Approximation by Superposition of Sigmoidal Functions. *Math Control Signal* 2:303-314
- DeVore RA (1998) Nonlinear approximation, *Acta Numerica* 8:51-150
- Elsner JB, Tsonis AA (1992) Nonlinear prediction, chaos, and noise. *B Am Meteorol Soc* 73:49-60
- Funahashi K (1989) On the Approximate Realization of Continuous Mappings by Neural Networks. *Neural Networks* 2:183-192
- Gell-Mann M, Lloyd S (1996) Information Measures, Effective Complexity, and Total Information. *Complexity* 2:44-52
- Hansen LK, Salamon P (1990) Neural network ensembles. *IEEE T Pattern Anal*, 12: 993-1001
- Hashem S (1997) Optimal linear combination of neural networks. *Neural Networks* 10:599-614
- Haupt SE, A Pasini, C Marzban (eds), (2009) *Artificial Intelligence Methods in Environmental Sciences.*, Springer-Verlag, New York
- Haykin S (2008) *Neural Networks and Learning Machines.* Pearson, NJ, USA
- Hornik K (1991) Approximation Capabilities of Multilayer Feedforward Network. *Neural Networks* 4:251-257
- Hornik K, Stinchcombe M, White H (1990) Universal approximation of an unknown mapping and its derivatives using multilayer feedforward network. *Neural Networks* 3:551-560
- Hsieh WW (2001) Nonlinear principal component analysis by neural networks. *Tellus* 53A:599-615
- Hsieh WW (2004) Nonlinear Multivariate and Time Series Analysis by Neural Network Methods. *Rev Geophys.* doi:10.1029/2002RG000112
- Hsieh WW (2009) *Machine Learning Methods in the Environmental Sciences.* Cambridge University Press, Cambridge
- Kon M, Plaskota L (2001) Complexity of Neural Network Approximation with limited information: a worst case approach. *J Complexity* 17:345-365
- Krasnopolsky VM (2007) Reducing Uncertainties in Neural Network Jacobians and Improving Accuracy of Neural Network Emulations with NN Ensemble Approaches. *Neural Networks* 20:454-461
- Krasnopolsky VM (2007) Neural Network Emulations for Complex Multidimensional Geophysical Mappings: Applications of Neural Network Techniques to Atmospheric and Oceanic Satellite Retrievals and Numerical Modeling. *Rev Geophys.* doi:10.1029/2006RG000200
- Krasnopolsky V.M., 2013, "The Application of Neural Networks in the Earth System Sciences. Neural Network Emulations for Complex Multidimensional Mappings", 200 pp., Springer
- Krasnopolsky VM, Fox-Rabinovitz MS (2006) Complex Hybrid Models Combining Deterministic and Machine Learning Components for Numerical Climate Modeling and Weather Prediction. *Neural Networks* 19:122-134

- Krasnopolsky VM, Kukulin VI (1977) A stochastic variational method for the few-body systems. *J Phys G Nucl Partic: Nucl Phys* 3:795-807
- Krasnopolsky VM, Chalikov DV, Tolman HL (2002) A neural network technique to improve computational efficiency of numerical oceanic models. *Ocean Model* 4:363-383
- Krasnopolsky VM, Gemmill WH, Breaker LC (1999) A multiparameter empirical ocean algorithm for SSM/I retrievals. *Can J Remote Sens* 25:486-503
- Krasnopolsky VM, Gemmill WH, Breaker LC (2000) A neural network multi-parameter algorithm SSM/I ocean retrievals: comparisons and validations. *Remote Sens Environ* 73:133-142
- Krasnopolsky VM, Lord SJ, Moorthi S, Spindler T (2009) How to Deal with Inhomogeneous Outputs and High Dimensionality of Neural Network Emulations of Model Physics in Numerical Climate and Weather Prediction Models. *Proc of International Joint Conference on Neural Networks*, Atlanta, Georgia, USA, June 14-19, 1668-1673
- Krasnopolsky V. M., M. S. Fox-Rabinovitz, P. J. Rasch, and Wang M. (2014). "[Fast NN Emulation of the Super-Parameterization in the Multi-scale Modeling Framework.](#)" NCEP Office Note No. 476.
- Lee JW, Oh J-H (1997) Hybrid learning of mapping and its Jacobian in multilayer neural networks. *Neural Comput* 9:937-958
- Liano K (1996) Robust Error Measure for Supervised Neural Network Learning with Outliers. *IEEE T Neural Networ* 7:246-250
- Luengo J, Garcia S, Herrera F (2010) A study on the use of imputation methods for experimentations with Radial Basis Function Network classifier handling missing attribute values: he good synergy between RBFNs and Event Covering method. *Neural Networks* 23:406-418
- Maas O, Boulanger J-P, Thiria S (2000) Use of neural networks for predictions using time series: Illustration with the *El Niño Southern oscillation* phenomenon. *Neurocomputing* 30:53-58
- MacKay DJC (1992) A practical Bayesian framework for back-propagation networks. *Neural Comput* 4:448-472
- Maclin R, Shavlik J (1995) Combining the predictions of multiple classifiers: using competitive learning to initialize neural networks. *Proc of the Eleventh International Conference on Artificial Intelligence*, Detroit, MI, 775-780
- McCulloch WS, Pitts W (1943) A logical calculus of the ideas immanent in neural nets. *B Math Bioph* 5:115-137
- Nabney IT (2002) *Netlab: Algorithms for pattern recognition*. Springer-Verlag, New York
- Naftaly U, Intrator N, Horn D (1997) Optimal ensemble averaging of neural networks. *Comput Neural Syst* 8:283-294
- Neal RM (1996) *Bayesian learning for neural networks*. Springer-Verlag, New York
- Nguyen D, Widrow B (1990) Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights. *Proc of International Joint Conference of Neural Networks*, June 17-21, San Diego, CA, USA, 3: 21-26

- Opitz D, Maclin R (1999) Popular ensemble methods: an empirical study. *J Artif Intell Res* 11:169-198
- Reitsma F (2001) Spatial Complexity. Master's Thesis, Auckland University, New Zealand
- Richman MB, Trafalis TB, Adrianto I (2009) Missing Data Imputation Through Machine Learning Algorithm. In *Artificial Intelligence Methods in Environmental Sciences*. SE Haupt, A Pasini, C Marzban (eds), Springer-Verlag, New York
- Rumelhart DE, Hinton GE, Williams RJ (1986) Learning internal representations by error propagation. In *Parallel Distributed Processing*, vol 1, DE Rumelhart, McClelland JL, PR Group (eds), MIT Press, Cambridge, Mass
- Sharkey AJC (1996) On combining artificial neural nets. *Connect Sci* 8:299-313
- Tang Y, Hsieh WW (2003) ENSO simulation and prediction in a hybrid coupled model with data assimilation. *J Meteorol Soc Jpn* 81:1-19
- Vann L, Hu Y (2002) A Neural Network Inversion System for Atmospheric Remote-Sensing Measurements. *Proc IEEE Instrumentation and Measurement Technology Conference*, vol 2, 1613-1615. doi:10.1109/IMTC.2002.1007201
- Vapnik VN (1995) *The Nature of Statistical Learning Theory*. Springer, New York
- Vapnik VN, Kotz S (2006) *Estimation of Dependences Based on Empirical Data (Information Science and Statistics)*. Springer, New York
- Weigend AS, Gershenfeld NA (1994) The future of time series: learning and understanding, 1-70. In *Time series prediction. Forecasting the future and understanding the past*, AS Weigend, NA Gershenfeld (eds), Addison-Wesley Publishing Company, Reading, MA
- Wessels LFA, Bernard E (1992) Avoiding false local minima by proper initialization of connections. *IEEE T Neural Networ* 3:899-905
- Zorita E, von Storch H (1999) A survey of statistical downscaling techniques. *J Climate* 2:2474-2489

3 The NN-TVS structure and data sets

In this section the structure and components of the NN-TVS system are documented. Also, the structure and formats of input, output, and ancillary data sets are described.

3.1 The structure of the NN-TVS system

The NN-TVS system is a combination of IDL and FORTRAN codes governed by UNIX and IDL scripts. The basic, gross structure of the system is shown in Fig. 3.1. For description of data files used by the system see Appendix 1. The full list of IDL routines constituting the NN-TVS is presented in Appendix 2.

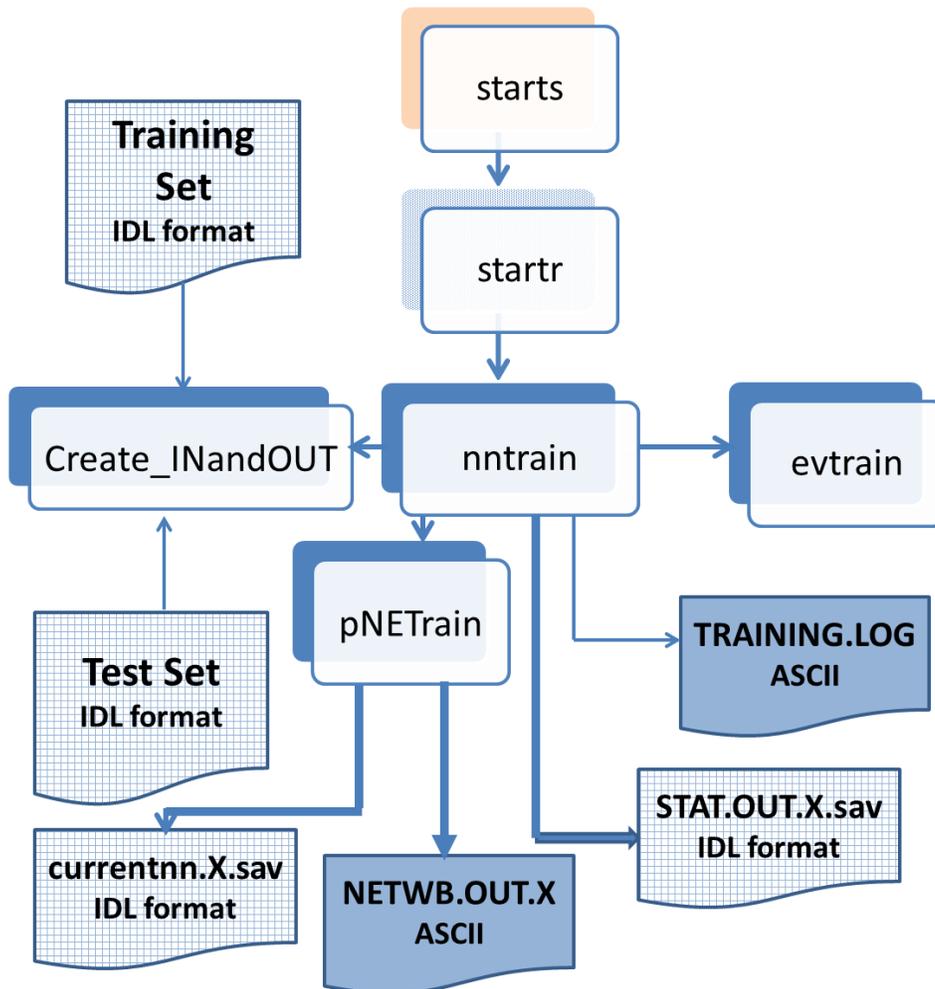


Figure 3.1. The gross structure of NN-TVS system. Color notation for scripts, codes, and data files: orange shade box – Unix script, light blue shade box – IDL script, white with the blue (solid) shade box – IDL code, blue box – data file in ASCII format, blue grid box – data file in IDL format. Thick arrows point to the output files.

3.2 Starting the system

To run the system (train NN), five files (scripts) have to be placed in the working directory (e.g., */Work*): *starts*, *startr*, *startfi*, *startfq*, and *qstart.sh*. UNIX script *starts* is used to start the system. It has four arguments and should be started as *./starts X b c d*. Here *X*, *b*, *c*, and *d* are the arguments. The first argument, *X*, shows the number of the training run, which can be seen in Figs. 3.1 and 3.2 as a part of data file names. Run number *X* ($0 \leq X \leq 99$) is used in all data file names and in the name of executable FORTRAN file *trainl.X* to distinguish different training runs. Table 3.1 shows possible values and the meaning of other arguments.

Table 3.1 Control parameters specified as the script *starts* arguments *b*, *c*, and *d*

Value/ Argument	0	1	2
<i>b</i>	Interactive training	Training in a queue	N/A
<i>c</i>	Restart	NW initialization	VB initialization
<i>d</i>	Calculation of Statistics only	Regular training	Restart FORTRAN code

Script *starts* calls an IDL script *startr* to start the system and to run main IDL program *nntrain.pro* (see Fig. 3.1 and 3.2). The text of the script is shown below with explanations introduced in red. It contains some control parameters that have to be modified for a particular problem and run. The presented script is written to run an example problem described below; however, for any other problem this script has to be modified. The parts of the scripts that have to be modified are shown in red. The control parameters are listed and explained in Table 3.2.

Table 3.2 Control parameters specified in the script *startr*

Parameter	Values	Meaning
dirc	String, UNIX directory	Location of the file <i>nntrain.pro</i>
dir	String, UNIX directory	Working directory
dird	String, UNIX directory	Location of the data files (training and test sets)
finr1	String, File name	Training set
fins	String, File name	Test set
in	2 to ~1,000	The number of NN inputs
hid	1 to ~500	The number of NN hidden neurons
out	1 to ~1,000	The number of NN outputs
Seed	Any integer	Seed for generator of random numbers
trmeth	0 or 1	0 - nonadjustable training (constant learning rates)

		1 - training with automatically adjustable learning rate
Maxnit	> 0	Maximal number of training iterations (epochs); an exit parameter
R1	$\sim 1.10^{-6} \leq R1 \leq \sim 1.$	Starting value of learning rate in the hidden layer; by default $R1 = 2 * R2$
R2	$\sim 1.10^{-6} \leq R2 \leq \sim 1.$	Starting value of learning rate in the hidden layer
EPS1	$\sim 1.10^{-4} \leq EPS1 \leq \sim 1.$	Minimal <i>RNE</i> (3.3); an exit parameter
EPS2	$\sim 1.10^{-4} \leq EPS2 \leq \sim 1.$	Minimal change in weights (3.1 and 3.2); an exit parameter
EPS3	$\sim 1.10^{-7} \leq EPS3 \leq \sim 1.10^{-4}$	Minimal change of <i>RNE</i> (3.3); an exit parameter
ITEX	> 1	Maximum number of iterations without decreasing error; an exit parameter
c1 & c2	Any, default is 1.	Constants reserved for outputs unit conversion
outscale[4,K]	$1 \leq K \leq \text{out}$	K – the number of types of outputs with different normalization
outscale_{0,i}	outj	Number of outputs of type j ; $0 \leq j \leq K-1$
outscale_{1,i}	notpj (0,1, or 2)	Scaling type for outputs of type j ; see eqs. (2.6) to (2.9)
outscale_{2,i}	$0. < AMA_j < 1.$	see eqs. (2.6) to (2.9)
outscale_{3,i}	cj - any	Constant for unit conversion of output of type j (see c1 and c2 above)
WW[out]	All > 0.	Optional weights for outputs in the error function eq. (2.11a) ¹

¹Introducing WW[out] corresponds to defining a matrix $\alpha_{ji} = WW[i]$ for all js in eq. (2.11a)

IDL script *start*

```

;*****
;
; This IDL file /script provides initialization parameters for automated
;   NN training & runs training program nntrain.pro
;   It is called by the UNIX script starts
;----
;-- All prints in this script directed to TRAIN.LOG.X ----
;----
;-----
;
runum = XXX ; run #, comes from starts
;
print,'# = ', XXX
;
; Set location of the core NN files; nntrain.pro & trainl
;
dirc = '/scratch1/portfolios/NCEPDEV/ocean/save/Vladimir.Krasnopolsky/Pro/NN-TVS/'
;
;-----
; --- Set Working and Data directories ---
;-----
;
;   Working directory
dir = '/scratch1/portfolios/NCEPDEV/ocean/save/Vladimir.Krasnopolsky/Pro/Work/'
;
;   Data directory
dird = '/scratch1/portfolios/NCEPDEV/ocean/save/Vladimir.Krasnopolsky/Pro/NN-TVS/'
;
;-----
;---- Set Training and Test file names:
;-----
;

```

```

finr1 = 'training_set.sav'      ; training set
;
finr1 = Dird + finr1
;
fins = 'test_set.sav'         ; test set
;
fins = Dird + fins
;
;-----
; --- Set NN Architecture and NN type
;-----
;
;      in - # of NN inputs
;      hid - # of neurons in the hidden layer
;      out - # of NN outputs
;-----
;
in = 69      ;      sdy,cdy,sla,slo,clo,64 x sst
;
hid = 2
;
out = 784
;
;-----
print,'IN = ',strtrim(in,2),'; HID = ',strtrim(hid,2),'; OUT = ',strtrim(out,2)
;
;-----
; --- Set the Method of NN Initialization, YYY comes from starts
;-----
;
;      imeth - method of initialization
;      if ' ' then initialization from the file NET.IO.X.sav,
;      where X = runum
;
imethod = YYY  ; 0 - from file Restart.IO.X.sav
               ; 1 - NW method
               ; 2 - VB method
;
if imethod eq 0 then imeth = '' else $
if imethod eq 1 then imeth = 'NGUEN-WIDROW' else $
if imethod eq 2 then imeth = 'VILLIERS-BARNARD'
;
print,'INITIALIZATION = ',imeth
;
;-----
; --- Seed - seed number for generator of random numbers
;      to generate initialization (e.g., 10001)
;-----
;
Seed = 110100I  This number has to be changed only if you want to train NN using the same initialization
; method but with different random numbers; different seeds can be used to train an ensemble of NNs.
;
;-----

```

```

; --- Set NN Training Method and Parameters
;-----
;
;   trmeth - 0 - nonadjustable training (constant learning rates)
;           1 - training with automatically adjustable learning rate
;
trmeth = 1
;
;   nit - max # of iterations
;   R1 - learning rate in hidden layer
;   R2 - learning rate in output layer
;
;-----
;
maxnit = 50000! ; default max number of training iterations
;
; zzz comes from starts
; --- if zzz = 0 then nit = -1! (to calculate statistics reading FORIDL.IO.X)
; --- if zzz = 1 then nit = maxnit perform a regular training
; --- if zzz = 2 then nit = -2! continue with FORTRAN
;
if (ZZZ eq 0) then nit = -1!
; calculate statistics reading FORIDL.IO.X
if ZZZ then nit = maxnit
; regular training
if (ZZZ eq 2) then nit = -2!
; continue training going to FORTRAN
;
PRINT, 'nit = ', strtrim(nit,2)
;
; --- Initialize the learning rates ---
R2 = 1.e-2 & R1 = 2. * R2  These numbers are more or less universal; however, they may be varied.
;
;-----
;
print,'MAX # of ITERATIONS = ',strtrim(nit,2),'; R2 = ',strtrim(r2,2)
;
;   EXIT PARAMETERS for FORTRAN program when learning rate
;   is adjusted automatically (trmeth=1):
;   EPS1 - MIN NORMALIZED RMSE
;   EPS2 - MIN CHANGE IN WEIGHTS
;   EPS3 - MIN CHANGE IN NORMALIZED RMSE
;   ITEX - MAX # OF ITERATIONS WITHOUT DECREASING RMSE  This number can be reduced to speed up
trainig
;   PRI - Priority of execution = 20 - PRI (0 =< PRI =< 19)
;   COM - 0 - interactive run; 1 - submit to queue
;
; --- QQQ comes from starts
MAXDFOR = {EPS1:0.01, EPS2:0.01, EPS3:0.00001, ITEX:500, PRI:0, COM:QQQ}
;
;-----
; Set output conversion coefficients
;-----

```

```

;
c1 = 1.
c2 = 1.
;
;-----
; --- Set the Output Scaling/Normalization
;-----
;
; outscale regulates the scaling of outputs
; outscale = fltarr[4,K]
; K - number of different types of the output
; SUM[j] outscale[0,j] = OUT
; outscale[1,j] = notp (0, 1, or 2)
; 0 - each output of this type is scaled in [-AMA,AMA] interval
; 1 - each i-th output is scaled as  $X'_i = AMA * (X_i - X_{i\text{mean}}) / SD_i$ 
;   or  $X' = AMA * (X - X_{\text{mean}}) / SD$ , where X, X', Xmean and SD are vectors
; 2 - all outputs of this type are scaled as  $X' = AMA * (X - X_{\text{mean}}) / SD$ 
;   where X and X' are vectors, Xmean and SD are total numbers
notp = 1
notp1 = 2
; outscale[2,j] = AMA - for above calculating
AMA = 0.1
; outscale[3,j] = C - conversion coefficient: c1, c2, etc. for use in evtrain.pro
;
out1 = 284
outscale = [[out-out1,notp,AMA,c1],[out1,notp1,AMA,c2]]
;
print,'OUTPUT NORMALIZATION: ', strtrim(outscale,2)
;
;-----
; --- Initialize Output Weights -----
;-----
;
; Error^2 = WW * (Y - Ynn)^2
;
WW= Make_Array(out,/float,value=1.) This setting may be modified to change weights of different outputs. See
also in Creat_INandOUT.pro for more complex weighting.
;
STW = 'WW[0:OUT-1] = 1.'
;
print, STW
;
;-----
; --- Run IDL Training code from NN-TVS directory
;-----
;
.r /scratch1/portfolios/NCEPDEV/ocean/save/Vladimir.Krasnopolsky/Pro/NN-TVS/nnttrain ; runs NN training
program
;
exit
;#####

```

Below some of the exit parameters defined in the script *startr* for the use in the FORTRAN code *trainl* are explained. First two parameters, *con1* (3.1) and *con2* (3.2), describe relative changes in NN weights in the hidden and output layers correspondingly. They are limited by parameter *EPS2* (see Table 3.2). The third parameter, *RNE* (3.3), is a normalized RMSE.

$$con1 = \sqrt{\frac{\sum_{ij} (\Delta W_{1ij})^2}{\sum_{ij} (W_{1ij})^2}} \quad (3.1)$$

$$con2 = \sqrt{\frac{\sum_{ij} (\Delta W_{2ij})^2}{\sum_{ij} (W_{2ij})^2}} \quad (3.2)$$

$$RNE = \sqrt{\frac{\sum_{ij} \alpha_{ij} (Y_{ij} - YN_{ij})^2}{\sum_{ij} \alpha_{ij} (Y_{ij})^2}} \quad (3.3)$$

3.3 Running the main program *nnttrain*, the structure of training sets

The script *startr* runs the IDL main program *nnttrain* shown in Figs. 3.1 and 3.2. This code (*nnttrain*) calls an IDL subroutine *Create_INandOUT.pro*, which reads the training set (some preprocessing of the sets may be added there). After reading the training set, *nnttrain.pro* calls *pnnttrain.pro* to perform NN training. The work of *pnnttrain.pro* is described in the next (3.4) section. After training has finished, *nnttrain.pro* calls *Create_INandOUT.pro* to read the test set and then calls *evnttrain.pro* to evaluate the NN performance on the training and test sets.

The training set is in the file **training_set.sav** and the test set in **test_set.sav**. Both files are in IDL format and have the same structure; they contain:

- info – string array with information about the data set and
- two 2D arrays *tsetin*[nr,in] and *tsetout*[nr,out]. The first array contains NN inputs and the second – NN outputs
 - nr is the number of records/patterns in the data set.

The results of the evaluation (evaluation statistics) are written in **STAT.OUT.X.sav** file in IDL format. This file contains:

- Info – string array, which describes the data in the file
- Outputs = {Y:y, YN:yn, Yt:yt, YNT:ynt} – an IDL structure containing:
 - y and yt – outputs (data) in training and test sets correspondingly
 - yn and ynt – NN outputs on training and test sets correspondingly
- Outstat = {trls:statrl, trps:statrp,tsls:statsl, tsps:statsp} – an IDL structure containing:
 - statr(s)l[0:OUT-1] – IDL structures arrays with statistics for each output (r) on training and (s) on test sets: $\text{statr(s)l}[i] = \text{BCORMS}(y(t)[*,i], yn(t)[*,i])$; see description of BCORMS
 - statr(s)p[K] – IDL structures arrays with statistics for each type of outputs, that are determined by array outscale.
- Outscale – array that defines types of outputs, which differ by the type of the scaling applied (see the text of the script *startr* above)
- Nnar = [in, hid, out] – array describing NN architecture (number of input, hidden neurons, and outputs)

The file **STAT.OUT.X.sav** can be used to print out and plot different statistics, using additional IDL programs. *nnttrain.pro* also writes some information in the ASCII file **TRAINING.LOG**, which keeps track of all training sessions in **Work** directory. See also Appendix 4 for explanations and definitions of some output statistics.

3.4 Running the program *pnetrain*

nnttrain.pro calls *pnetrain.pro* to perform NN training. The basic structure of the IDL subroutine *pnetrain.pro* is shown in Figs. 3.3 and 3.4. Execution of *pnetrain.pro* starts from saving the run number X in the file **RUNUM.IF**. Then the restart status is saved in the file **RESTATUS.X** (0 for the restart and 9 for a new training). Next, the NN inputs are scaled; they are converted to an interval [-1,1]. The outputs are also scaled in accordance with the scaling information given in the array outscale defined in the IDL script *startr* (see Sects. 3.2 and 2.2.3).

The scaling of outputs is followed by initialization of NN weights using either subroutine *nwgtini* or *wgtinti* (see Sect. 2.3) depending on the value of the third argument of the script *starts* (see Table 3.1). If the run X is a restart run, then the initial values of the NN weights are read by the FORTRAN program *trainl* from the file **Restart.IO.X**.

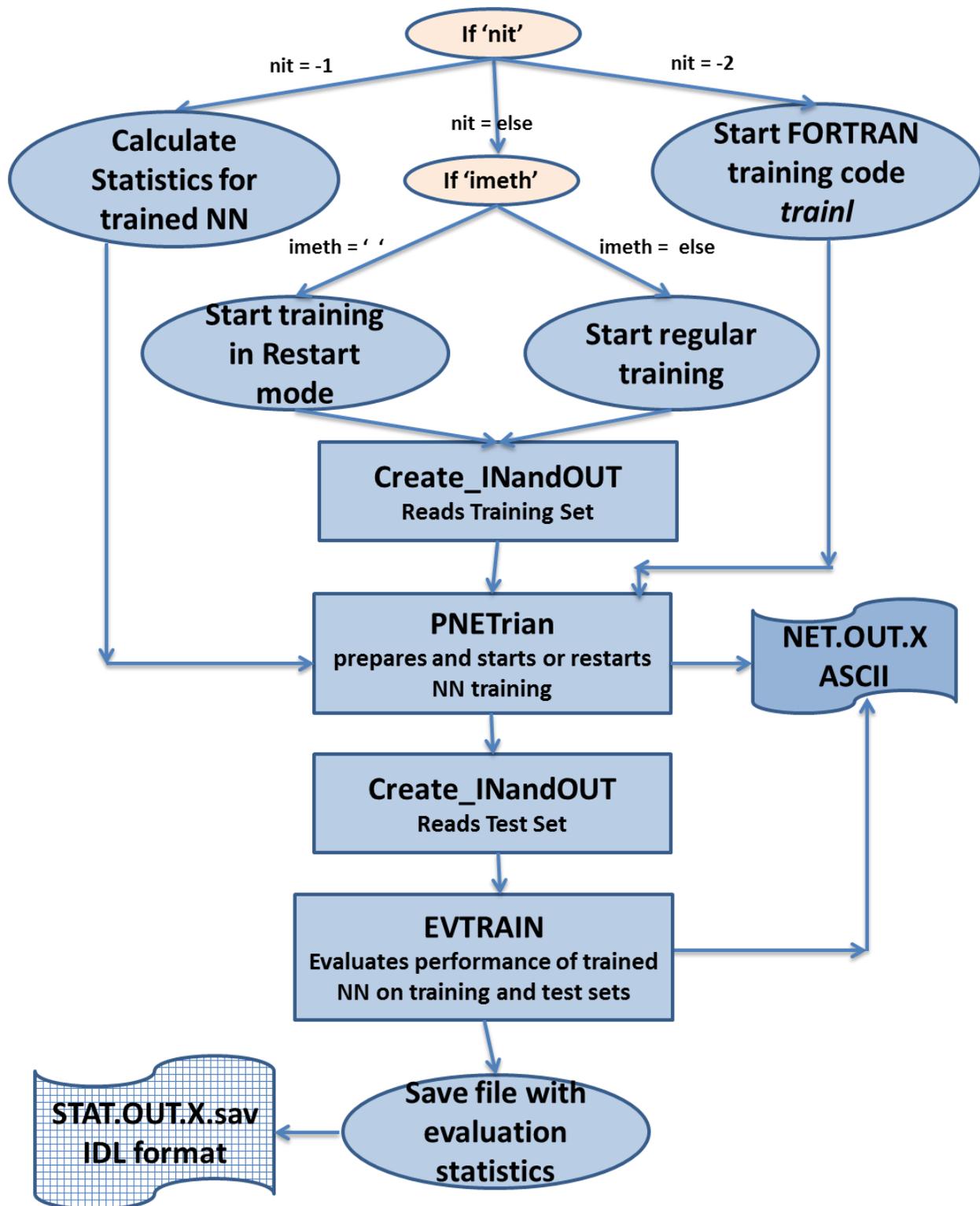


Figure 3.2. The structure chart of *nntrain.pro*

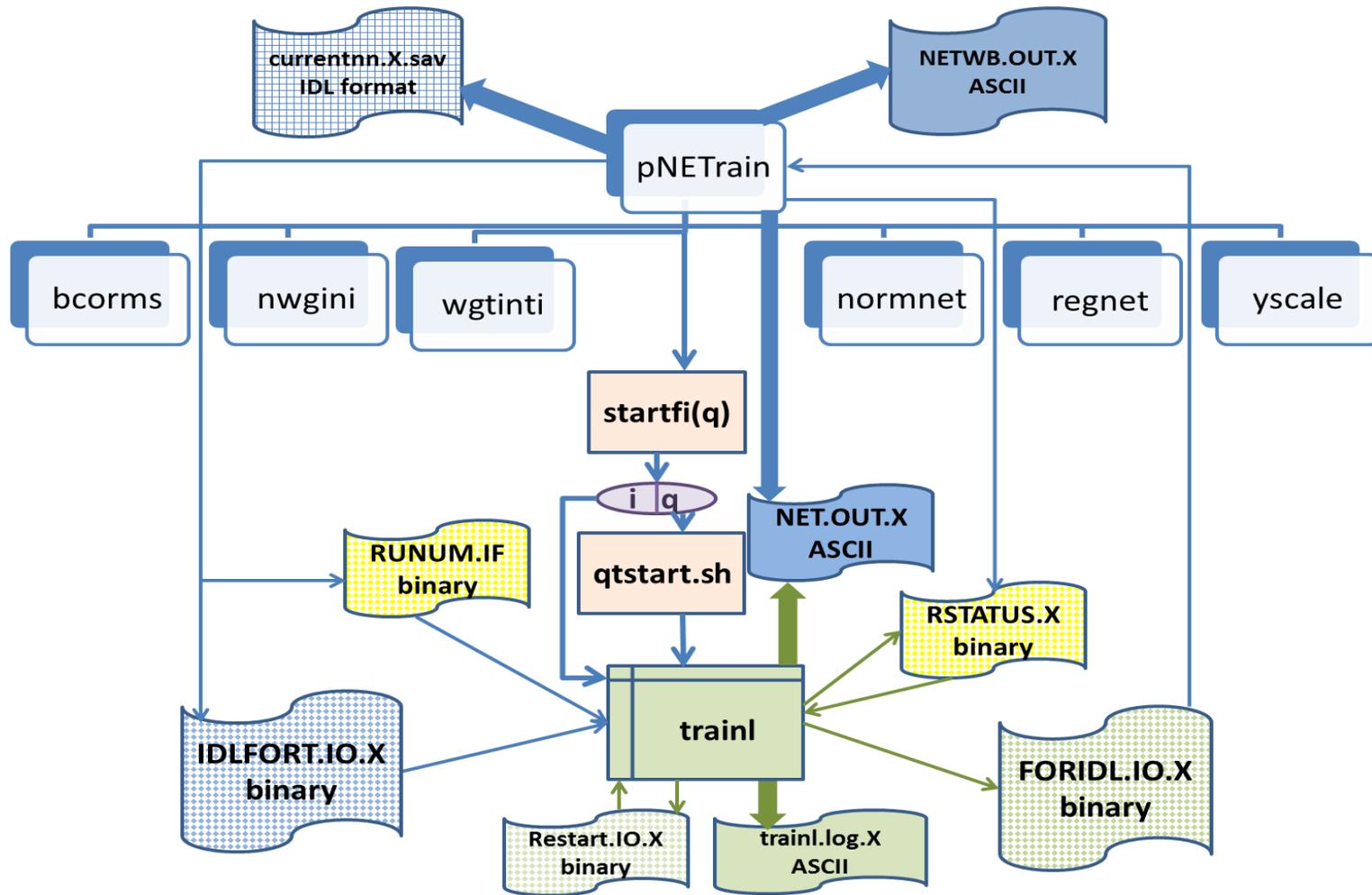


Figure 3.3 The basic structure of *pnetrain.pro*. Color notation for scripts, codes, and data files: orange box – Linux script, blue shade box – IDL code, green box – FORTRAN code, solid wavy box – data file in ASCII format (blue – written by IDL code, green – by FORTRAN code), blue wavy grid box – data file in IDL format, blue wavy small squares box – data file in binary format (written by: blue – IDL code, green – FORTRAN code), and yellow wavy box – ancillary data file in binary format.

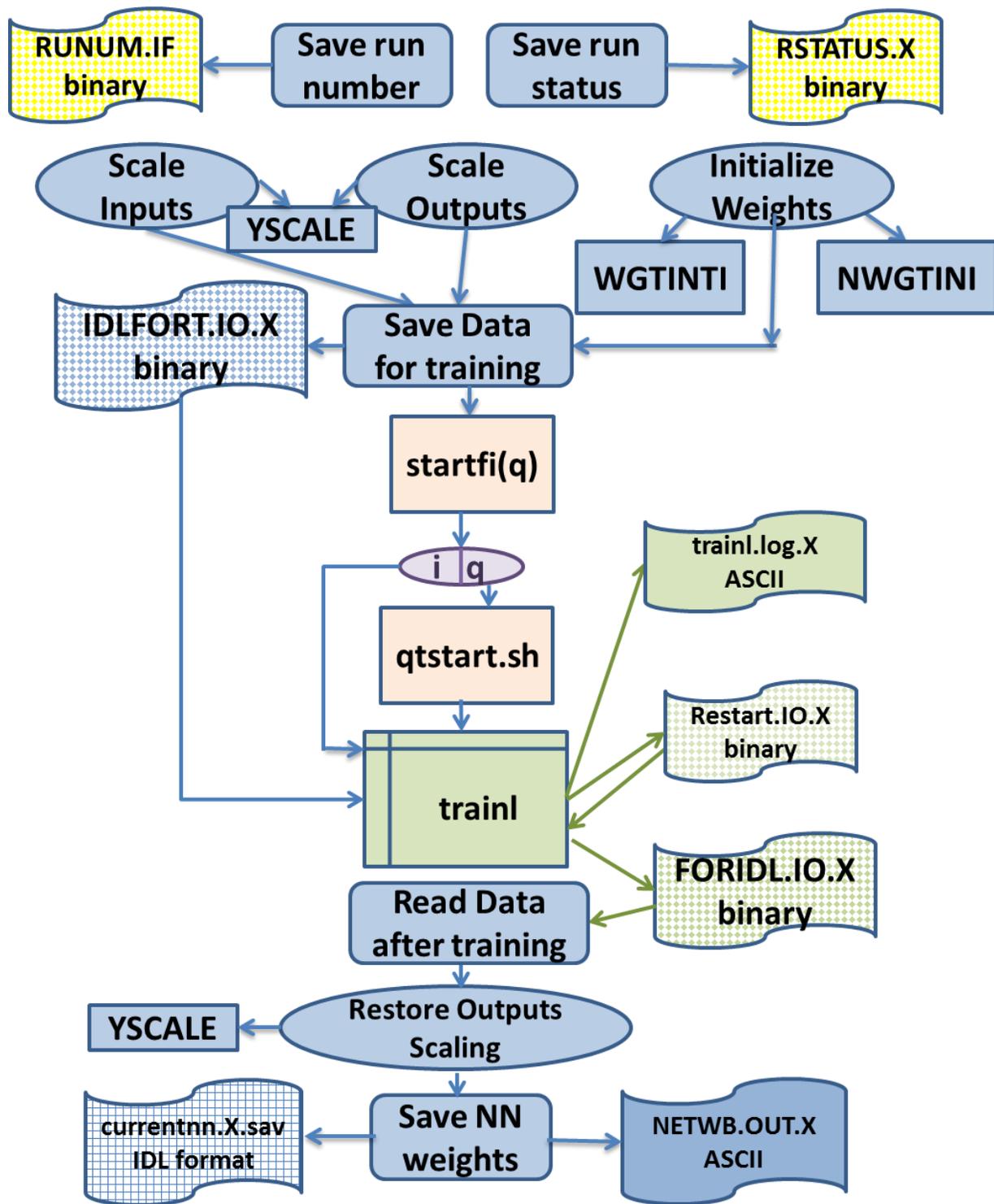


Figure 3.4 Data flow in the subroutine *pnetrain.pro*. See also caption to Fig. 3.3

Subsequently, all information required for the NN training, including scaled NN inputs and outputs, initial NN weights, matrix of output weighting parameters, and multiple control parameters are saved in the file **IDLFORT.IO.X** to be read by the FORTRAN program *trainl*.

After the file **IDLFORT.IO.X** is saved, the UNIX script *startfq* (or *startfi*) is run from IDL to send the FORTRAN program *trainl* in the queue (*startfq*) or start training on an interactive node (*startfi*). The FORTRAN program *trainl* (see next Sect. 3.5) performs NN training per se (minimization of error function). Periodically it saves the state of minimization in the binary file **Restart.IO.X**. After the training is completed, *trainl* saves all training results, including found NN weights, in the binary file **FORIDL.IO.X**. If the training session is running on the front end node¹ the control is returned to the IDL code *pnetrain*. If the training session is running on computing node, the IDL code has to be restarted as *./starts X 0 0 0*. It reads the file **FORIDL.IO.X**, restores the output scaling and writes the outputs files with NN weights: the IDL file **currentnn.X.sav** and the ASCII file **NETWB.OUT.X**.

3.5 Restarting NN training

If training is interrupted because of any reason, and both files **IDLFORT.IO.X** and **Restart.IO.X** are still in Work directory, the run X can be restarted as *./starts X 0 0 2*, which restarts the FORTRAN program *trainl*. If **IDLFORT.IO.X** is not available any more, the run X can be restarted as *./starts X 0 0 1*. This command restarts the *nntain* to recreate **IDLFORT.IO.X** and then restarts the FORTRAN main program *trainl*. If both files **IDLFORT.IO.X** and **Restart.IO.X** are lost, the training should be started from scratch (see Sect. 3.2)

3.6 Running the FORTRAN code *trainl*

FORTRAN code *trainl.f90* is located in subdirectory */NN-TVS/For/* together with the script *go*, which allows compiling this code. The data flow in *trainl* is shown in Fig. 3.5. This program performs sequential (or on-line) NN training (see Sect. 2.3.1). After starting, first the program reads file **RUNUM.IF** to determine the run number, X. Next it reads **RSTATUS.X** to determine if this run is a new training or a restart run. Then it reads **IDLFORT.IO.X** file, and if this is a

¹ Using Zeus computer is assumed here.

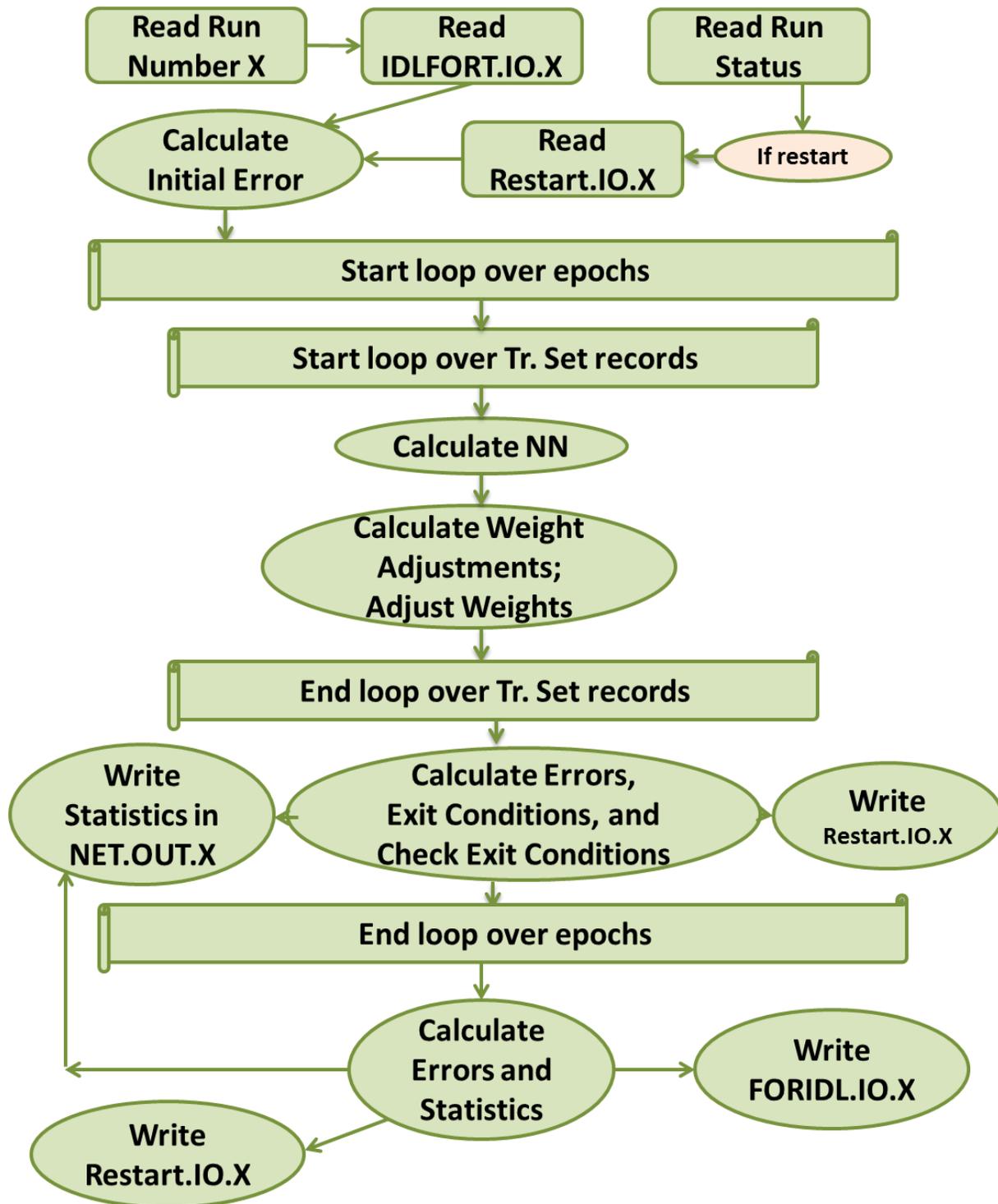


Figure 3.5 Data flow in the FORTRAN code *trainl*.

restart run, it reads **Restart.IO.X**. Subsequently, it calculates initial value of the error function running the NN with initial weights on the training set.

After calculating initial error, the loop over the number of epoch starts. Each epoch corresponds to the one time pass over the all records of the training set. The number of epochs is limited by the parameter `maxnit`; 50,000 is default max number of training iterations set in the script `stsrtr` (see Sect. 3.2, eqs. (3.1 – 3.3), and Table 3.3). Inside the epoch loop the loop over the records of the training set starts. For each record, the square error ξ_i^2 (see eq. 2.11) is calculated. It is used to calculate NN weight adjustments and then adjust NN weight before moving to the next record. Thus, sequential (or on-line) training (see Sect. 2.3.1) is implemented here.

After an epoch is completed, errors are calculated, exit conditions are checked (see Table 3.3), and if they are satisfied, the training is completed. Otherwise, the next training epoch starts. Every ten training epochs the restart file **Restart.IO.X** is saved, and every p training epochs ($p = 10$ if the number of training epoch is less than 1,000, and then $p = 100$) the error statistics is written in the file **NET.OUT.X**. If training is completed, the restart file **Restart.IO.X** and file **FORIDL.IO.X**, which contains complete information about training results, are saved.

Table 3.3 Exit conditions for NN training

Condition number	1	2	3	4
Condition	$RNE < EPS1$	$ITCOUNT^1 > ITEX$	$con1 < EPS2$ and $con2 < EPS2$	$\Delta RNE < EPS3$

¹ITCOUNT is the number of iterations without error improving

3.7 Returning to the IDL codes *pnetrain* and *nnetrain*

If the training is running on a front end node, the control returns to *pnetrain* automatically; otherwise, the command `/starts X 0 0 0` has to be issued in the working directory. IDL code reads the file **FORIDL.IO.X**, it also calls the IDL subroutine `YSCALE` to restore the original scaling of the NN outputs, and writes the NN weights in the IDL file **currentnn.X.sav** and in ASCII file **NETWB.OUT.X**. Subsequently, control returns to *nnetrain*, which calls the IDL subroutine `Create_INandOUT` to read the test set, and then it calls the IDL subroutine `EVTRAIN` to evaluate the performance of the trained NN on training and test sets. The results of the evaluation (evaluation statistics) are written in **STAT.OUT.X.sav** file in IDL format (see Sect. 3.3 for details). The evaluation statistics is also written in the file **NET.OUT.X**. **This completes the NN training run.**

4 Using trained NN

After a NN or an ensemble of NNs has been trained, it can be used to emulate the original mapping (2.1). All information required to build and calculate the NN emulation is saved in the file **NETWB.OUT.X**. For an NN ensemble emulation there are multiple files **NETWB.OUT.X_i** ($i = 1, \dots, N$), where N is the number of the ensemble members. The FORTRAN module `neural.f90` contains the necessary code to calculate the NN emulation or the NN ensemble emulation. This module is located in the same NN-TVS/Fort directory as the NN training code `trainl.f90`. The module contains three FORTRAN subroutines:

1. Subroutine `init_nn_emulator` reads file(s) **NETWB.OUT.X_i** and initializes NN(s) architecture and weights. It must be called before the first call of the `nn_emulation` subroutine
2. Subroutine `nn_emulation` calculates the actual emulation of the original mapping. Provided with proper inputs it generates the mapping outputs. Using provided inputs, it creates the vector of NN inputs (which may be a subset of the mapping inputs), calls the subroutine `compute_nn` and creates the mapping output vector.
3. Subroutine `compute_nn` given the vector of inputs calculates the NN or the NN ensemble outputs.

The full text of the module `neural.f90` is presented in Appendix 3.

5 Conclusions

This Note describes the NN training and validation system or NN-TVS developed at NCEP (EMC) during last two decades. This system evolved following the evolution of problems to solve. There exist various commercial and shareware softwares for training NNs; however, usually the major intent of developers of such software is to develop universal software capable of solving as many generic problems as possible. As was mentioned in Sect. 1, the majority of oceanic and atmospheric applications, from the mathematical point of view, can be formulated as complex, multidimensional, nonlinear mappings and, theoretically speaking, can be solved with a generic type of NN – the Multilayer Perceptron. MLP is usually incorporated in almost all aforementioned NN softwares. Unfortunately, these NN softwares cannot be use because of some specific features of atmospheric and oceanic problems.

For example, special features of our applications include:

1. High dimensionality of input and output vectors
2. Outputs are usually physically related and correlated
3. Complexity and strong nonlinearity of Input-to-Output relationship

This features lead to high complexity of the NN and high dimensionality of training space. High dimensionality of the training space requires the training set of a large size to sample this space. Both high complexity of the NN and the large size of the training set lead to a long NN training time. Thus, the NN training has to be run on high performance computers. The NN-TVS has been developed to target our applications and to be run on our computers. Because of aforementioned specificities of atmospheric and oceanic application multiple limitations have been introduced in the design of the NN-TVS to make the NN training affordable in terms of the training time and the memory required. For example, the MLP architecture has been limited; the MLP NN with only one hidden layer and linear output layer has been implemented. Also, only the mostly robust training algorithm – back-propagation training – has been implemented. In our opinion, a wide variety of NN applications developed using the NN-TVS (Krasnopolsky 2013) justify the principles underlying the design of the NN-TVS.

6 Appendixes

Appendix 1. Data files used by NN-TVS

Table A.1 shows all data files created and used by NN-TVS in the process of the NN training and validation. First two files are located in the data directory specified in the script *startr*. All other files are located in the directory *Work/*.

Table A. 1 Data files created and used by NN-TVS and (except first two) residing in directory *Work/*

#	File name	Format	Comments
1	training_set.sav	IDL binary	See Sect. 3.3for description of the file structure
2	test_set.sav	IDL binary	--- “ ---
3	RUNUM.IF	Binary	Contain number of the run (X)
4	RSATATUS.X	Binary	Shows status of the run # X (regular or restart)
5	IDLFORT.IO.X	Binary	Transfers information necessary for NN training from IDL nntain.pro to FORTRAN code trainl
6	FORIDL.IO.X	Binary	Transfers information after NN training from FORTRAN code trainl to IDL nntain.pro
7	Restart.IO.X	Binary	Saves information necessary to restart training run # X
8	NET.OUT.X	ASCII	Contains important information about training run # X
9	TRAIN.LOG.X	ASCII	Contains full information about training run # X
10	trainl.log.X	ASCII	Contains information about the running FORTRAN code trainl for run # X (included in NET.OUT.X)
11	STAT.OUT.X.sav	IDL binary	Statistics on training and test set for NN # X
12	currentnn.X.sav	IDL binary	Weights of the trained NN (run # X)
13	NETWB.OUT.X	ASCII	Weights of the trained NN (run # X)

Appendix 2. List of IDL and FORTRAN functions and subroutines used by NN-TVS

Table A2 contains the list of IDL and FORTRAN functions and subroutines used by NN-TVS.

Table A2 List of functions and subroutines used by NN-TVS

#	Name	Language	Function
1	create_INandOUT	IDL subroutine	Reads data and creates NN INs & OUTs for training & testing
2	BCORMS	IDL function	Calculates multiple statistics
3	prstate	IDL subroutine	Prints output of BCORMS
4	YSCALE	IDL function	Scales and unscales the NN output vector Y or a part of Y
5	EVTRAIN	IDL subroutine	Evaluates performance of trained NN on training and test sets
6	REGNET	IDL function	Calculates a neural network output vector in original coordinates.
7	DREGNET	IDL function	Calculates neural network outputs in orig. coordinates + derivatives of outputs over inputs (Jacobian). Included for future use.
8	NORMNET	IDL function	Calculates a neural network output vector in normalized coordinates.
9	WGTINTI	IDL subroutine	Initializes weights of NN using Wessels and Bernard algorithm
10	NWGTINI	IDL subroutine	Initializes weights of NN using Nguen-Widrow algorithm
11	pNETRAIN	IDL subroutine	Prepares everything for NN training and starts it
12	nntrain	Main IDL prog.	Main program that manages pre-processing, training, and validation
13	trainl	Main FORTRAN prog.	Performs NN training

Appendix 3. FORTRAN module neural.f90, which implements trained NN

After NN was trained and NN weights have been determined, this NN can be applied using the equations (2.2) and (2.3). The following FORTRAN module implements a trained NN in FORTRAN. This is a generic code. The parts of code that have to be changed for a particular problem are shown in red.

```
!*****
!  
! Name: neural  
!  
! Language: FORTRAN           Type - MODULE  
!  
! Version: 1.0      Date: 10-12-10   Written by: A. Belochitski and V. Krasnopolsky  
! Version: 2.0      Date: 06-12-14  
!  
!*****  
!  
! Module contains all subroutines required to initialize and  
! calculate ensemble of NNs.  
!  
!*****  
!  
    module neural  
    implicit none  
    private  
    public :: init_nn_emulator, nn_emulation  
!  
! Number of members in the NN ensemble  
!  
    integer, parameter :: nn_num_of_members = 3  
!  
! Files containing NN weights and biases  
!  
    character(*), parameter::nn_file_name(nn_num_of_members)= &  
        (/scratch1/portfolios/NCEPDEV/ocean/save/Vladimir.Krasnopolsky/Pro/Work/NETWB.OUT.0', &  
        /scratch1/portfolios/NCEPDEV/ocean/save/Vladimir.Krasnopolsky/Pro/Work/NETWB.OUT.1', &  
        /scratch1/portfolios/NCEPDEV/ocean/save/Vladimir.Krasnopolsky/Pro/Work/NETWB.OUT.2'/)  
!  
! Internal types and variables  
!  
    type nndata_1d  
        real, allocatable :: a(:)  
    end type nndata_1d  
!  
    type nndata_2d  
        real, allocatable :: a(:, :)  
    end type nndata_2d  
!  
! NN Hidden and output weights  
    type(nndata_2d) :: nn_w1(nn_num_of_members),nn_w2(nn_num_of_members)  
! NN Hidden and output biases  
    type(nndata_1d) :: nn_b1(nn_num_of_members),nn_b2(nn_num_of_members)  
! NN Number of inputs, hidden neurons and outputs  
    integer :: nn_in(nn_num_of_members),nn_hid(nn_num_of_members), &  
        nn_out(nn_num_of_members)  
!  
    contains  
!  
! Initialize NNs  
!  
    subroutine init_nn_emulator(me)  
!  
        integer, intent(in) :: me  
        integer iin,ihid,iout,member  
!  
        if (me == 0) print*, 'Module NEURAL: Number of NN ensemble members:', &
```

```

        nn_num_of_members
!
! Load NNs weights and biases
!
    do member=1,nn_num_of_members
!
        open(unit=1,err=10,file=trim(nn_file_name(member)),status='old')
        read(1,'(3i5)') iin,ihid,iout
        nn_in(member)=iin; nn_out(member)=iout; nn_hid(member)=ihid
!
        allocate(nn_w1(member)%a(iin,ihid),nn_w2(member)%a(ihid,iout))
        allocate(nn_b1(member)%a(ihid),nn_b2(member)%a(iout))
!
        read(1,*,err=11,end=12) nn_w1(member)%a, nn_w2(member)%a, &
            nn_b1(member)%a, nn_b2(member)%a
        close(1)
!
        if (me == 0) print*, 'Module NEURAL: NN File Loaded: ', &
            nn_file_name(member)
!
    end do
!
! All NNs in the ensemble must have the same number inputs and outputs
!
    if (.not.all(nn_in == nn_in(1)).or..not.all(nn_out == nn_out(1))) then
        if (me == 0) print *, "Module NEURAL: MME NN ensemble members have different number of inputs and/or outputs. Exiting."
        stop
    endif
!
    return
!
! Catch file opening/reading errors
!
10    if (me == 0) print *, "Module NEURAL: Error opening file ", &
        nn_file_name(member), ". Exiting."
    stop
!
11    if (me == 0) print *, "Module NEURAL: Error reading file ", &
        nn_file_name(member), ". Exiting."
    stop
!
12    if (me == 0) print *, "Module NEURAL: Reached EOF too early ", &
        nn_file_name(member), ". Exiting."
    stop
!
end subroutine init_nn_emulator
!
! NN emulator
!
subroutine nn_emulation(                &
! Inputs:
&    xin,&
! Outputs:
&    yout)
!
! NN Emulation
!
! Inputs
    real, intent(in):: xin(:)
!
    real :: rlat,rlon,cmc,cmcglb,dwd,emcwf,ghs,rjma,mam,ukmo,rmedley,mme
!
! Local variables
    real, parameter :: rmiss = -999.0
    real :: xx
!
    real :: nn_input_vector(nn_in(1)), nn_output_vector(nn_out(1))
!
    integer :: k
!

```

```

real, intent(out):: yout(:)
!
  nn_input_vector(1) =  xin(...)
  nn_input_vector(2) =  ...
  nn_input_vector(3) =  ...
  nn_input_vector(4) =  ...
  .....
!
  call compute_nn(nn_input_vector,nn_output_vector,nn_num_of_members,&
    nn_w1,nn_w2,nn_b1,nn_b2,nn_hid)
!
  yout   = nn_output_vector
!
end subroutine nn_emulation
!
subroutine compute_nn(X,Y,num_of_members,w1,w2,b1,b2,nhid)
!
! Input:
!   X(IN) NN input vector
!   integer, intent(in) :: num_of_members,nhid(num_of_members)
!   real, intent(in)::X(:)
!   type(nndata_2d), intent(in) :: w1(num_of_members),w2(num_of_members)
!   type(nndata_1d), intent(in) :: b1(num_of_members),b2(num_of_members)
!
! Ouput:
!   Y(OUT) NN output vector
!   real, intent(out):: Y(:)
!
! Local variables
!   integer i, nout
!   real, allocatable :: x2(:),x3(:)
!   integer member
!
!   nout=size(Y)
!
!   Y = 0.
!
!   allocate(x3(nout))
!
!   do member = 1,num_of_members
!
!     allocate(x2(nhid(member)))
!
! Calculate neurons in the hidden layer
!
!     forall(i = 1:nhid(member)) x2(i)= tanh(sum(X*w1(member)%a(:,i)+ &
!       b1(member)%a(i))
!
! Calculate NN output
!
!     forall(i=1:nout) x3(i)= sum(w2(member)%a(:,i)*x2) + b2(member)%a(i)
!
!
!     Y = Y + x3
!
!     deallocate(x2)
!
!   end do          ! member
!
!   deallocate(x3)
!
!   Y = Y / num_of_members
!
end subroutine compute_nn
!
end module neural

```

Appendix 4. Output Statistics calculated by NN-TVS

For each type of outputs NN-TVS calculates and writes in the file **NET.OUT.X** a set of statistics, which looks like (description of statistics are presented in red):

1 – OUTPUT IN tr.set, 2 - NN OUTPUT type #1 – index 1 indicates output data taken from the validation data set and statistics for these data;
 index 2 indicates NN output data and statistics.

SAMPLE SIZE = 28753580 28753580 28753580 – N1 - number of finite elements in data taken from the validation data set (finite means not NaN), N2 - number of finite elements in NN output data, and N3 - number of finite elements in their difference.

MIN1 = -1.710e+00 MAX1 = 3.390e+01 MED1 = 2.115e+01 – MIN1(2), MAX1(2), MED1(2) are min, max and median values of data sets
 MIN2 = -9.460e-01 MAX2 = 3.074e+01 MED2 = 2.142e+01 1 and 2 correspondingly

MEAN1 = 1.914e+01 STD1 = 8.365e+00 – mean value and standard deviation of data sets 1 and 2 correspondingly
 MEAN2 = 1.915e+01 STD2 = 8.304e+00

MAD1 = 7.239e+00 SKW1 = -5.197e-01 Kur1 = -9.975e-01 – mean absolute deviation (see eq. A4.1), skewness and kurtosis of data sets 1
 MAD2 = 7.251e+00 SKW2 = -5.237e-01 Kur2 = -1.077e+00 and 2 correspondingly

Statistics for Set1 – Set2 difference

1 - 2 STATISTICS:

=====

DMIN = -1.326e+01 DMAX = 9.675e+00 – min and max differences

BIAS	RMS	STD	CORC.	R2	S	SI	– mean difference (BIAS), RMS difference, standard deviation of the difference - (STD), correlation coefficient (CORC), the coefficient of multiple determination (R2) assuming that set 2 is the approximation and set 1 is the data (see eq. A4.4), skill score (S, see eq. A4.5), scatter index (SI = RMS/MEAN1)
1.204e-02	7.346e-01	7.345e-01	0.996	0.992	0.992	0.038	
MED	MAE	MAD	MAPE(%)	SKW	KURT		– median of the difference (MED), mean absolute error (MAE see eq. A4.1), mean absolute deviation of the difference (see eq. A4.2), mean absolute percentage error (MAPE see eq. A4.3), SKW and KURT are skewness and kurtosis of the difference
-0.001	5.281e-01	5.282e-01	8.042e+00	-0.173	7.001		

Mean absolute error (difference):

$$MAE = \frac{1}{N} \sum_{i=1}^N |set1_i - set2_i| \quad (A4.1)$$

Mean absolute deviation:

$$MAD1 = \frac{1}{N} \sum_{i=1}^N |set1_i - MEAN1| \quad (A4.2)$$

Mean absolute percentage error:

$$MAPE = \frac{100}{N} \sum_{i=1}^N \frac{(set1_i - set2_i)}{set1_i} \quad (A4.3)$$

Coefficient of multiple determinations, R2, assuming that set 2 is the approximation and set 1 is the data:

$$R2 = 1 - \frac{\sum_{i=1}^N (set1_i - set2_i)^2}{\sum_{i=1}^N (set1_i - MEAN1)^2} \quad (A4.4)$$

Skill score:

$$S = \frac{4 \cdot (1 + CORC)^4}{den}$$

$$den = \left(sig + \frac{1}{sig} \right)^2 \cdot (1 + R0)^4 \quad (A4.5)$$
$$sig = \frac{STD2}{STD1}$$

R0 is the maximum *CORC* attainable (here it is set to 1.).