Rijksuniversiteit Groningen Faculteit der Wiskunde en Natuurwetenschappen

Vakgroep
Informatica

Exploiting Network Redundancy for Low-Cost Neural Network Realizations

Hessel Keegstra

begeleiders: J.T. Udding
J.A.G. Nijhuis

april 1996

RuG

Contents

Abstract.

Neural networks are constructed and trained on powerful workstations. For real world applications, however, neural networks need to be implemented on devices (e.g. embedded controllers) with limited precision, storage and computational power. The mapping of a trained ideal neural network to such a limited environment is by no means straightforward. One has to consider proper word size selection and activation function simplification, without disturbing the network behavior too much (i.e. both networks have to meet a user specified specification). This transformation process reduces the available redundancy in a trained network without affecting it's behavior.

In this thesis, we present two redundancy reduction approaches which automatically determine and exploit the available redundancy in a trained neural network to transform the network into a simplified version (with approximately the same behavior) which can be implemented with less cost. The simplification procedures address the activation function type as well as the selection of the required weight precision and range. This also includes pruning of unnecessary connections and neurons. The usefulness of the presented approaches is illustrated by an image processing and an optical character recognition (OCR) application.

The first approach is a greedy algorithm which selects a neuron (or connection) and tries to replace it with a less expensive one. After a change, the algorithm checks to see if the specifications are still met. This method is slow and it usually does not yields an efficient result, however, it always guarantees a behavioral invariant transformation.

A faster transformation is achieved by the local approach. This method tries to pinpoint the redundancy of a neuron with data local to that neuron, i.e. input and output swing. Based on the local data, redundancy indices are calculated which are used to determine the activation function replacement. Weight precision selection is performed by a bit–pruning procedure; it uses connection swing or a Karnin connection sensitivity analysis to determine the required number of bits of a connection. The local approach is fast and delivers efficient results, however, the transformations are not independent of each other. This means, a behavioral invariant transformation can not always be guaranteed.

A combination of the global and the local approach is suggested. The redundancy indices can be used to guide the optimization procedure of the global approach. This ensures a behavioral invariant transformation, delivering a sub–optimal neural network.

Samenvatting.

Neurale netwerken worden gebouwd en getraind op krachtige workstations. In veel industriële applicaties zullen neurale netwerken geïmplementeerd moeten worden op systemen (bv. embedded controllers) met beperkte precisie, geheugenruimte en rekenkracht. Het afbeelden van een getraind neuraal netwerk op een systeem met deze beperkingen is niet eenvoudig. Er moet rekening gehouden worden met het vervangen van activatiefuncties en het bepalen van de optimale woordbreedte zonder dat het netwerk een ander gedrag krijgt. Het transformatie proces reduceert de in een getraind neural netwerk aanwezige redundantie, zonder dat dit het gedrag van het netwerk beïnvloet.

In deze scriptie worden twee redundantie reductie methoden voorgesteld. Beide methoden bepalen automatisch de redundantie in een getraind neuraal netwerk, om vervolgens het netwerk te converteren in een netwerk met hetzelfde gedrag, welke gemakkelijker te implementeren is. Zowel de activatiefunctie als de woordgrootte (aantal bits) worden per neuron (of connectie) bepaald. Een niet functionele connectie of neuron wordt gewijderd (gepruned). De twee procedures worden gedemonsteerd aan de hand van een beeldbewerking en een optical character recognition (OCR) probleem.

De eerste aanpak is een zogenaamd greedy algoritme. Deze globale procedure neemt een neuron (of connectie) en probeert deze te vervangen door een goedkoper alternatief. Na iedere verandering controleert het algoritme of het netwerk nog aan de specificaties voldoet. De methode is vrij langzaam en levert niet altijd een efficiënt resultaat op; de transformatie echter altijd gedrags invariant.

Een snellere transformatie biedt de lokale methode. Deze methode probeert de redundantie in een neuron te bepalen aan de hand van data welke voor dat neuron lokaal beschikbaar is. Met deze data (input- en outputswings) worden redundantie indices bepaald, welke gebruikt worden om een geschikte vervanging van de activatie functie te bepalen. De woordgrootte selectie wordt uitgevoerd met een bit-pruning methode. Bit-pruning maakt, om te bepalen hoeveel bits er voor een connectie nodig zijn, gebruik van de connectie swing of de Karnin connectie gevoeligheidsanalyse. De lokale aanpak is snel en levert een efficiënt resultaat op. Het nadeel is dat de transformaties niet onderling onafhankelijk zijn. Dit betekent dat de lokale aanpak niet altijd een gedrags invariante transformatie uitvoert.

Een combinatie van beide methoden biedt meer perspectieven. Met behulp van de redundantie indices wordt het optimalisatie proces van de globale methode beter gestuurd. Dit garandeert een gedrags invariante transformatie welke een efficiënt resultaat op kan leveren.

Chapter 1

Neural Network Basics

1.1 Introduction

Work on artificial neural networks, commonly referred to as *neural networks* has been motivated right from its inception by the recognition that the brain computes in an entirely different way from the conventional digital computer. The brain is a highly *complex, nonlinear, and parallel computer.* It has the capability of organizing neurons so as to perform certain computations (e.g. pattern recognition and perception) many times faster than the fastest digital computer. In building artificial neural networks (ANNs) one attempts to capture some of the brain's learning, generalization and fault–tolerance capabilities [1], [2].

The use of neural networks offers the following useful properties and capabilities:

- *Nonlinearity*
  A neuron is basically a nonlinear device. Consequently, a neural network, made up of an interconnection of neurons, is itself nonlinear. Nonlinearity is a highly important property, since most physical mechanisms responsible for the generation of an input signal (e.g. speech signals) are nonlinear of nature.
- *Adaptivity*
  Neural networks have a built–in capability to *adapt* their synaptic weights to changes in the surrounding environment. A neural network trained to operate in a specific environment may be easily *retrained* to deal with minor changes in the operating environmental conditions.
- *Fault tolerance*
  If a single neuron or its interconnections are damaged, recall of knowledge is impaired in quality. However, the knowledge encapsulated in a neural network is distributed over the network. Therefore the damage has to be extensive before the overall response of the network is degraded seriously. Thus, in principle, a neural network exhibits a graceful degradation of performance rather than catastrophic failure.

- *VLSI implementability*
  The massively parallel nature of neural networks makes it potential fast for the computation of certain tasks. This same feature makes a neural network ideally suitable for *very large scale integration* (VLSI) technology.

In trying to understand the functionality of the brain, the idea of *neurons* as basic elements was introduced. Neural networks are composed of a large amount of interconnected neurons. The manner in which the neurons of a neural network are structured is intimately linked with the learning algorithm used to train the network. In general, we may identify three different classes of network architectures:

1. Single–Layer feedforward networks
A *layered* neural network is a network of neurons organized in the form of layers. In the simplest form of a layered network, we just have an *input layer* of source nodes that projects onto an *output layer* of neurons (figure 1.1a). The input layer is merely a placeholder, the actual computation is performed in the following layer (hence the name single–layer).

2. Multilayer feedforward networks
The second class of a feedforward neural network distinguishes itself by the presence of one or more *hidden layers*, whose computation nodes are correspondingly called *hidden neurons* or *hidden units* (figure 1.1b). The function of the hidden units is to intervene between the external input and the network output. By adding one or more hidden layers, the network is enabled to extract higher–order statistics, for the network acquires a *global* perspective despite its local connectivity by virtue of the extra set of synaptic connections and the extra dimension of neural interactions.

3. Recurrent networks
A *recurrent neural network* distinguishes itself from a feedforward neural network in that it has at least one *feedback* loop. For example, a recurrent network may consist of a single layer of neurons with each neuron feeding its output signal back to the inputs of all the other neurons, as illustrated in figure 1.1c. The presence of feedback connections has a profound impact on the learning capabilities of the network, and on its performance. Moreover, the feedback loops involve the use of particular branches composed of *unit–delay elements*, denoted by z^{-1}. In this thesis we limit ourselves to feedforward neural networks.

Each network architecture can have a local–, a nearest neighbor or a fully connected connection scheme. Figure 1.1 shows only the fully connected connection scheme.

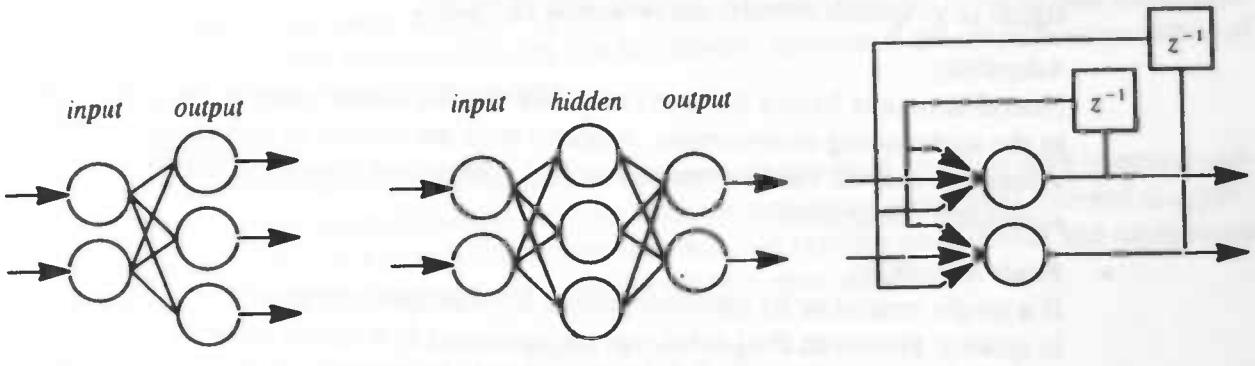


Figure 1.1: *a) Single–Layer Feedforward Network. b) Multilayer Feedforward Network. c) Recurrent Network*

1.2 Model of a Neuron

A *neuron* is an information–processing unit that is fundamental to the operation of a neural network [1], [2]. Figure 1.2 shows the *model* for a neuron. We may identify three basic elements of the neuron model, as described here:

- A set of *synapses* or *connection links*, each of which is characterized by a *weight* or *strength* of its own. Specifically, a signal x_j at the input of connection j connected to neuron k is multiplied by the synaptic weight w_{kj}. It is important to make a note of the manner in which the subscripts of the synaptic weight w_{kj} are written. The first subscript refers to the neuron in question and the second subscript refers to the input end of the synapse to which the weight refers. The weight w_{kj} is positive if the associated synapse is excitatory; it is negative if the synapse is inhibitory.
- An *adder* or *summing junction* for summing the input signals, weighted by the synapses of the neuron. This operation constitutes a *linear combiner.*
- An *activation function* for selective amplification of the weighted sum and limiting the amplitude of the output of a neuron. The activation function is also referred to in the literature as a *transfer, squashing* or *limiting function* in that it squashes (limits) the amplitude range of the output signal to some finite value. Typically, the normalized amplitude range of the output of a neuron is [0,1] or [–1,1].

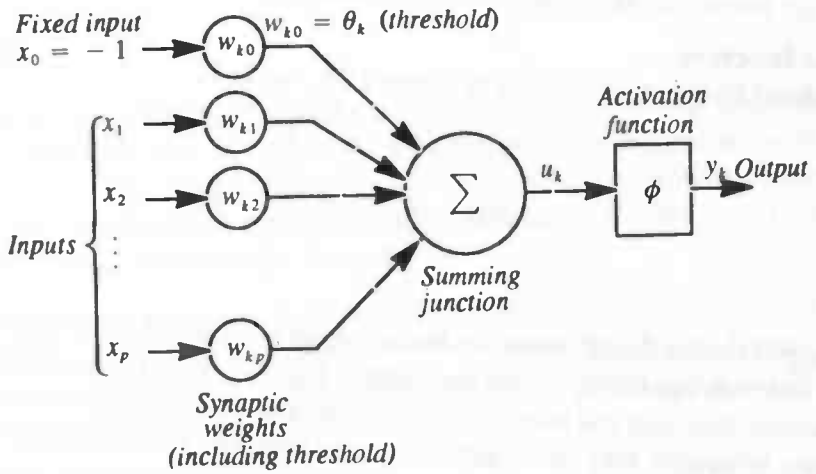


Figure 1.2: *Model of a neuron*

In mathematical terms, we may describe a neuron k by the following pair of equations:

$$u_k = \sum_{j=1}^{p} w_{kj} x_j - \theta_k \tag{1.1}$$

and

$$y_k = \phi(u_k) \tag{1.2}$$

where $x_1, x_2, \ldots, x_p$ are input signals; $w_{k1}, w_{k2}, \ldots, w_{kp}$ are the synaptic weights of neuron k; u_k is the *linear combiner output;* θ_k is the *threshold;* ϕ is the *activation function*; and y_k is the output signal of the neuron. The use of the threshold θ_k has the effect of applying an *affine transforma-*

tion (shift) to the output u_k of the linear combiner in the model of figure 1.2. Alternatively, we may model the threshold as a bias, i.e. a positive input connected to a negative weight. However, both models are mathematically equivalent.

The threshold (or bias) can be modelled as a constant input to the linear combiner, we may rewrite equation 1.1 as:

$$u_k = \sum_{j=0}^{p} w_{kj} x_j$$

where $w_{k0} = \theta_k$ and $x_0 = -1$.

The activation function, denoted by ϕ, defines the output of a neuron in terms of the activation level at its input. We may identify three basic types of activation functions:

1. Hardlimiter function.
For this type of activation function, described in figure 1.3c, we have

$$\phi(v) = \begin{cases} 1 & , v \geq 0 \\ 0 & , v < 0 \end{cases} \tag{1.3}$$

Some authors also call this type of activation function a threshold functions. It is sometimes desirable to have the activation function range from –1 to +1. Binary valued neurons with this type of activation functions are called *Adalines* (adaptive linear element). Neural networks composed of Adalines are known as *Madelines* (multiple–adaline).

2. Threshold function.
For the threshold function, described in figure 1.3b, we have

$$\phi(v) = \begin{cases} 1 & , v \geq \frac{1}{2} \\ v & , -\frac{1}{2} > v < \frac{1}{2} \\ 0 & , v \leq -\frac{1}{2} \end{cases} \tag{1.4}$$

where the amplification factor inside the linear region is assumed to be unity (a similar definition exists for a function operating within the range –1 to +1). This form of activation function is a *piecewise–linear* function and may be viewed as an approximation to a nonlinear amplifier. The following two situations may be viewed as special forms of the piecewise function: **1.** A *linear combiner* results if the linear region of operation is maintained without running into saturation. **2.** The piecewise–linear function reduces to a *hardlimiter function* if the amplification factor of the linear region is made infinitely large.

3. Sigmoid function.
The sigmoid function is by far the most common form of activation function used in neural networks. Neurons using this particular type of activation functions are sometimes referred to as *perceptrons*. It is defined as a strictly increasing function that exhibits smoothness and asymptotic properties. An example of the sigmoid is the *logistic function*, defined by:

$$\phi(v) = \frac{1}{1 + e^{(-av)}} \tag{1.5}$$

where a is the *slope parameter* of the sigmoid function. Figure 1.3a shows the *logistic function* with various different slopes (depending on the choice of a). Sometimes it is useful to use a sym-

metrical version (i.e. odd function) of the logistic function, defined as:

$$\phi(v) = \tanh(v) \tag{1.6}$$

There is a close relation between the logistic function and its symmetrical version:

$$\tanh(v) = \frac{\sinh(v)}{\cosh(v)} = \frac{e^{v} - e^{-v}}{e^{v} + e^{-v}} = 2\phi_{logistic}(2v) - 1 \tag{1.7}$$

Sigmoid functions are differentiable, whereas threshold function are not. Differentiability is an important feature for network learning, as will be described in the next section.

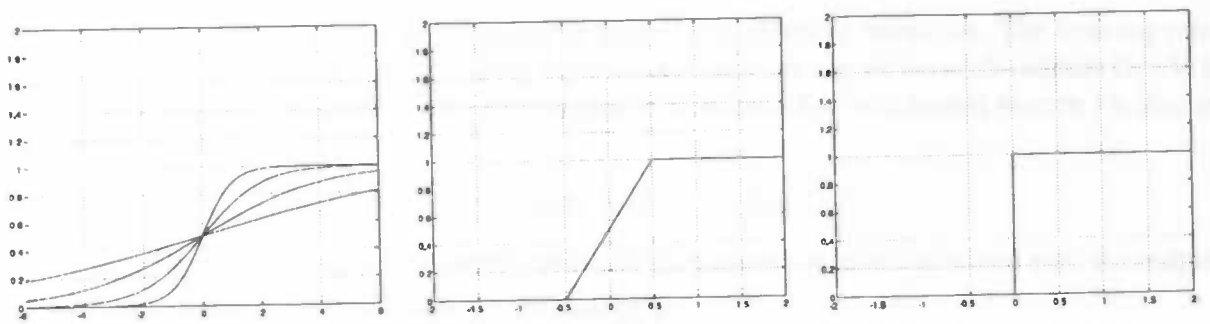


Figure 1.3: *Commonly used activation functions: a) sigmoid functions with different slopes, b) threshold function, c) hardlimiter function*

1.3 Learning

One of the major advantages of neural networks is the capability of learning. Learning is a process by which the free parameters (i.e. the networks weights, etc.) of a neural network are adapted through a continuing process of stimulation by the environment in which the network is embedded. Learning can even be achieved when an explicit relationship between input and output data in unknown or difficult to describe.

In this section, we will focus on a special class of neural networks called *multilayer perceptrons* (MLPs). These have been applied successfully to solve some difficult and diverse problems by training them in a *supervised* manner with a highly popular algorithm known as the *error back–propagation algorithm* (or simply *back–prop* or BP).

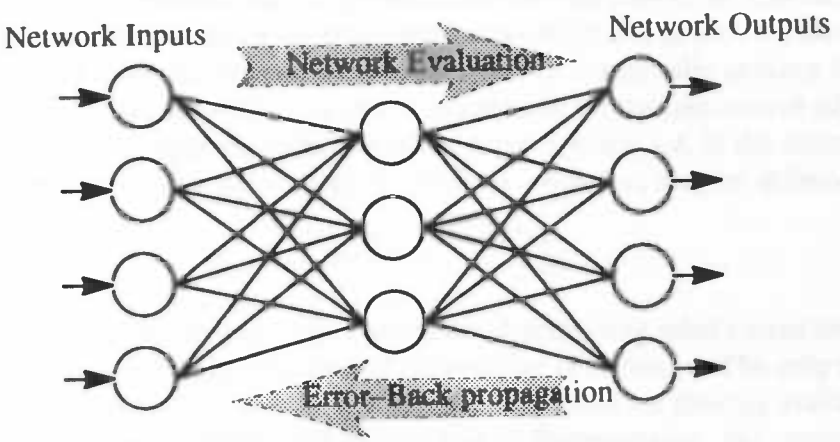


Figure 1.4: *Schematic view of the forward and backward pass.*

Basically, the error back–propagation process consists of two passes: a forward pass and a backward pass (as shown in figure 1.4). In the *forward pass,* the network evaluation takes place. An input vector is applied to the input nodes of the network, and its effect propagates through the network, layer by layer. During this pass the synaptic weights remain constant. During the *backward pass*, on the other hand, the synaptic weights are all adjusted. The actual response of the network is subtracted from the desired (target) response to produce an *error signal,* as shown in (1.8). The desired response is applied by an *external teacher* (hence the name *supervised*). The error signal is then propagated backward through the network, against the direction of the synaptic connections (hence the name error back–propagation). The synaptic weights are adjusted so as to make the error signal smaller.

During the training phase, the network is trained a number of iterations. The training process stops when the network outputs satisfy a given specification, e.g. all network outputs should have an error smaller than some ϵ. The error signal at iteration n for each output neuron j is thus computed as follows:

$$e_j(n) = d_j(n) - y_j(n) \tag{1.8}$$

where $d_j(n)$ denotes the *desired response value* for neuron j at iteration n, and $y_j(n)$ the output response of neuron j at iteration n, as defined in (1.2).

A commonly used measure for the output error of a neural network is the summed squared error over all outputs, which is defined as follows :

$$\mathcal{E}(n) = \frac{1}{2}\sum_{j\in C} e_j(n)^2 \tag{1.9}$$

where C denotes all neurons in the output layer of the network.

This represents a *cost–function*, and the objective in the learning process is to minimize this cost–function. The only way we can minimize the error is to adjust the network's free parameters, which are the interconnection weights (and, naturally, the thresholds (or biases) of the neurons, but they may be viewed as weights too). We can say that the error of the network is dependent on the weight values present in the network. Thus, the learning of a neural network may be viewed as the search for the optimal weight values.

One way of finding the desired minimum in weight space is by using the method of *steepest descent.* According to this method, the weights have a time–varying form, and their values are adjusted in an iterating manner along the error–surface with the aim of moving them progressively towards the optimal solution. This method has the task of continually seeking the bottom point of the error–surface of the network. It is intuitively reasonable that successive adjustments to the weights have to be in the opposite direction of the error–surface, i.e. in the direction opposite to the gradient $\partial\mathcal{E}(n)/\partial w_{ji}(n)$. The adjustment $\Delta w_{ji}(n)$ for a weight w_{ji} may be defined as follows:

$$\Delta w_{ji} = -\eta\frac{\partial\mathcal{E}(n)}{\partial w_{ji}(n)} \tag{1.10}$$

The symbol η denotes the *learning rate parameter*, determining what extent the weight adjustment should follow the opposite direction of the gradient $\partial\mathcal{E}(n)/\partial w_{ji}(n)$. The only thing to do now, is to express the gradient $\partial\mathcal{E}(n)/\partial w_{ji}(n)$ in network identities that are directly available, such as the activation functions, error–signals and output–levels. Summarizing, the correction $\Delta w_{ji}(n)$ applied to the synaptic weight connecting any neuron i to neuron j is defined by the *delta–rule*:

$$\begin{bmatrix} Weight \\ correction \\ \Delta w_{ji}(n) \end{bmatrix} = \begin{bmatrix} learningrate \\ parameter \\ \eta \end{bmatrix} \begin{bmatrix} local \\ gradient \\ \partial_j(n) \end{bmatrix} \begin{bmatrix} input\ signal \\ of\ neuron\ j \\ y_i(n) \end{bmatrix} \tag{1.11}$$

The error back–propagation algorithm is the most popular training algorithm for MLPs. It is efficient in training but requires a large precision to ensure proper error convergence as it involves a relatively large number of computations (iterations) in the forward and the backward passes. Error back–propagation requires about 16 to 21 bits precision [21]; the actual amount is problem depend. Therefore, error back–propagation is difficult to implement in hardware [21]–[25]. Usually, back–propagation training is performed on a powerful workstation with fast floating–point hardware.

For a closer and more theoretical look at the error back–propagation algorithm and a complete description of other learning strategies, the reader is referred to [1] and [2].

1.4 Neural Network Construction

To exploit the full benefits of artificial neural networks is not as easy as it seems. The design of a neural network has become a tedious trial–and–error process with complicated dependencies between the system parameters. A neural network approach needs a basic set of three functions: *network topology, learning rule,* and *a set of training samples*. Because of the application domain dependence and interdependencies between these three functions, an optimal combination is hard to find. It is hardly possible to tune a single network parameter without affecting a dozen others in a rather unpredictable way.

The selection of a certain network topology is determined by the application area of the neural network. Feedforward topologies have successfully been applied in pattern recognition related problems, whereas feedback (recurrent) networks are used mostly in optimization problems and associative memories. The network topology strongly influences the network training rule and training strategy.

The required number of external inputs and outputs is not as easily established as it might seem. For most problems the number of inputs and outputs is not clearly defined. For example, control applications for dynamical systems require not only actual sensor data but also previous sensor data, resulting in an unknown number of additional inputs. A closer look at the available data is also necessary. Just applying all data directly to a neural network will almost certainly result in a failure, since input data often contains redundant or even conflicting information.

To overcome these problems, the real–world data need to be preprocessed to remove redundant information. During this preprocessing phase, important *features,* which characterize the important information present in the data set, are extracted, whereas redundant information is eliminated. Without this dimensional reduction, the amount of information would be too much for the network, preventing the network from discovering the important information during training. It is clear that the usage of a priori knowledge is very important since it influences the overall performance of the neural network.

Another important design issue is the number of hidden layers, the number of neurons located in each hidden layer and the number of interconnections. The network must be large enough (i.e. contain enough free parameters) to be able to "express" the function to be learned. If the network is too small, the function can not be learned (underfitting). Too many hidden neurons usually results in a poor generalization characteristic (overfitting). Determining the right network size is

a trial–and–error game. Though research has been performed in this area [4]–[6], no clear guidelines are available yet.

A small network is very sensitive to initial conditions and local minima in the error–surface, but the reduced complexity favors improved generalization. On the other hand, a large network is less sensitive and through that easier to train. An approach called *pruning* is to train a network that is larger than necessary and then remove the parts that are not needed. The large initial size allows the network to learn reasonably quickly with less sensitivity while the trimmed system offers improved generalization. Many pruning algorithms can be put into two broad groups. One group estimates the sensitivity of the error function to the removal of an element [17], [18]; the element with the smallest effect can then be removed. The other group adds penalty terms to the cost–function (1.9) that rewards the network for choosing efficient solutions [17],[19],[20]. In general, the sensitivity methods modify a trained network (i.e. after training), and the penalty terms methods use the error back–propagation algorithm to modify (i.e. set unnecessary weights to zero) the network during training.

In close relation with the network size is the number of training examples. Generalization is influenced by the size and efficiency of the training set and the structure of the network. The larger the network, the more training samples are needed. Equation (1.12) states a common rule of thump:

$$N > \frac{W}{\epsilon} \tag{1.12}$$

where N is the number of training samples, W is the total number of synaptic weights and ϵ denotes the fraction of error permitted on the test set. Thus, with an error of 10 percent, say, the number of training examples should be approximately 10 times the number of synaptic weights in the network. If the training set is to small the network can not learn the function (underfitting), whereas a training set which is too large will result in a poor generalization characteristic. The presence of conflicts in the data set may also prevent a neural network from learning a function properly. Conflicts are patterns in which the inputs are nearly the same, but the outputs are quite different. These conflicts may be due to erroneous measurements or poor features (*underdimensioning*) which do not contain relevant information. It is clear that the training examples must be carefully selected from the available real–world data.

1.5 Fault Tolerance

Fault tolerance is the neural networks ability to perform well in the presence of errors. In a neural network, fault tolerant behavior is obtained by the distributed storage of the network's internal knowledge at all the connections in the network.

Two types of fault tolerance can be distinguished [3]:

- *Data fault tolerance*
  The networks ability to handle disturbed input data correctly. Data fault tolerance is usually measured as the maximum Hamming distance between an input pattern and the corresponding training patters, whereby correct classification is still guaranteed.
- *Hardware fault tolerance*
  The networks ability to correctly compute its function, despite hardware that is in an erroneous state as the result of failures of components, e.g. broken connections or connections with an erroneous weight or neurons with an incorrect activation function.

Fault tolerance is often considered as an intrinsic characteristic of neural networks, this is not entirely true [3]. Data fault tolerance can be enhanced by adding a noise component to the training set. However, extending the training time results in a better distribution of the network's internal knowledge, and so enhancing hardware fault tolerance. Simply extending the number of hidden units does not always result in a better hardware fault tolerance. As a consequence fault tolerance can be trained and must be considered as a design specification.

Hardware fault tolerance is a consequence of *redundancy* [9],[11],[15], i.e. additional "resources" incorporated into a system with the aim of masking the effects of faults. In [7] and [8] a well known technique called TMR (triple modular redundancy, a sort of majority vote) is used to extend the network hardware fault tolerance.

In literature several methods are advocated to measure the network redundancy or fault tolerance. In [8] redundancy is measured using SVD (singular value decomposition) analysis of the weight matrix. Others use statistical tools to investigate the performance (fault tolerance) degradation caused by stuck–at faults in VLSI implementations of neural networks. Most of these theories have little use in situations where a µ–processor is used to simulate the neural network.

There is a difference between the level of redundancy available during the training phase and during the recall phase. During training a large number of bits are required and the activation function is used over a wide domain. At the recall phase such a large precision is not needed and the domain over which the activation function is used is fixed and much smaller. As a consequence, there is more redundancy in a neural network during recall.

1.6 Mapping

Neural networks are currently realized in a number of ways. As we have seen, they can be implemented as computer simulations on a powerful workstation. For real–world applications, however, neural networks need to be implemented on devices (e.g. embedded controllers) with limited precision, storage and computational power.

Neural networks can be realized in software on µ–processors, DSPs or µ–controllers. For higher speed, neural networks can be implemented on analog, digital (e.g. ASICs) or hybrid (analog/digital) hardware [1]–[3]. No matter the choice of the target technology, mapping a neural network is not as easy as it looks.

The realization of the non–linear activation function can be problematic. Neurons with a hardlimiter or threshold activation function can be easily realized. The usual sigmoid transfer function, however, requires an analog approach, making exact fit and reproductions complicated. The realization of the dynamic range and resolution of the connection weights (and threshold) is another problem. Most target technologies do not offer infinite precision and range, so compromises have to be made regarding both properties. Experiments [16] are needed to determine to what extent the realized functionality differs from the functionality specified by the neural function, and whether these differences can be tolerated or not.

Neural network training requires large weight and activation function precision, as well as differentiable (sigmoidal) activation functions. This requirement makes on–chip learning difficult and expensive (large chip size) to implement. Because of the limited precision environment, on–chip learning can also be very difficult [21]–[25] in usage.

As noted earlier, not all precision and dynamic range is needed by the recall phase. During recall a large amount of redundancy is present in the network. Instead of manually trying to optimize

a solution, a better approach would to construct an algorithm (heuristic) which determines and exploits the redundancy in a trained ideal neural network to transforms this network into a simplified version (with approximately the same behavior) which can be implemented with smaller cost. This simplification algorithm should address the activation function type as well as the selection of the required weight precision and range.

1.7 The Problem

In this thesis we discuss an automatic simplification process (see figure 1.5) which determines and exploits the redundancy in a network to transform a trained ideal neural network into a (sub–)optimal neural network, utilizing reduced weight precision and simpler activation function types (i.e. non–sigmoidal). Further, unnecessary connections and neurons are determined and subsequently removed from the network. The simplification algorithm has to be behavior invariant, i.e. both networks are operating within a user–specification.

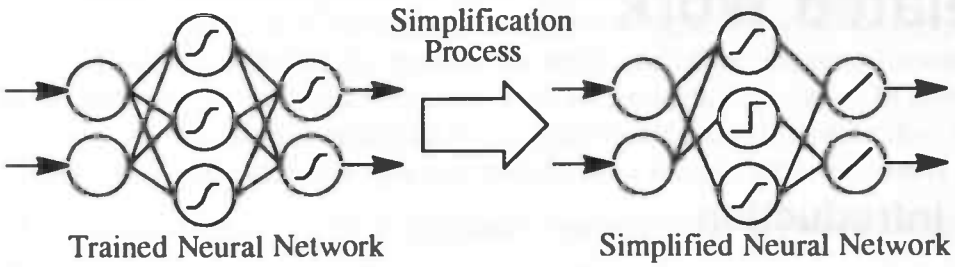


Figure 1.5: *Simplification Process*

The simplification process has to meet the following five functional specifications:

- The simplification process has to ensure that the ideal and the simplified neural network are both operating within a user–specification, i.e. it has to be behavior invariant.
- If possible, the sigmoid activation functions are replaced by simpler activation functions that are cheaper to evaluate.
- The required number of bits of each connection is determined and set accordingly.
- Irrelevant connections and neurons are determined and removed (pruned).

After this optimization phase the mapping is much easier. The implementation of the sub–optimal neural network is guaranteed to have the same behavior as the ideal neural network.

Chapter 2

Related Work

2.1 Introduction

The design parameter sensitivity of a neural network is an important consideration for hardware realizations. In this section we present in more or less detail some papers addressing sensitivity analysis and weight accuracy selection methods.

The theory and methods presented in this chapter do not consider selecting alternative activation functions. Weight accuracy selection is based on sensitivity analysis that analyses the finite word width effect on an ensemble of neural networks. We on the other hand, try to optimize particular trained neural networks by addressing weight accuracy selection as well as selecting alterative activation functions.

2.2 General

The sensitivity of feedforward layered networks of Madaline to weight and input errors is studied in [26]. An approximation is derived for the error probability (output inversion) of an output neuron as a function of the percentage weight change in large networks. The compete binary environment provide a nice proof method based on n–dimensional geometry (hyperspheres) to model the network behavior. The authors demonstrate and prove that the error probability increases with the number of layers in the network and with the change percentage in the weights. Surprisingly, the error probability is essentially independent of the number of weights per neuron and of the number of neurons per layer, as long as these numbers are large ($\geq$ 100).

In [27] and [28] this theory is extended to MLPs. Sigmoid activation functions are quantized (i.e., table–lookup). The sensitivity analysis is carried out by evaluating the mean and variance of the induced errors. In a companion paper [29], the impact of finite precision in a VLSI implementation of a neural network for an image processing problem is studied.

Another statistical sensitivity analysis is demonstrated in [30]. In this paper the weight sensitivity in a single–output MLP with differentiable activation functions is presented. Formulas are

derived to compute the sensitivity arising from additive/multiplicative weight or input perturbations. Errors induced in the activation function, e.g. activation function quantization errors, are not considered.

An analysis of the quantization effect in a MLP using a statistical model is presented in [31]. General statistical formulas are developed as a function of a parameter called *effective non–linearity coefficient*. In [35] a technique is proposed for reducing the effect of quantization by weight scaling.

In the following two sections, we present in more detail a promising theory concerning a weight accuracy selection procedure based on a noise–to–signal ratio, and a finite precision analysis of neural network hardware realizations.

2.3 Noise to Signal Ratio Approach

In [33] and [34] simple analytical expressions, based on a stochastic neuron model, are presented for the variance of the output error of a multilayer neural network. Based upon these expressions, a technique is developed for selecting the appropriate weight accuracy in a neural network hardware realization. Because, in most cases, the hardware architecture is designed to allow implementations of many different neural networks (i.e. programmable weights), the effect of weight errors in an *ensemble of networks with differing weights* over a set of input vectors is taken.

The output of node n in layer l of an ideal sigmoidal neuron (perceptron) may be stochastically modelled by:

$$Y_{l,n} = \tanh(X_l^t W_{l,n}) \tag{2.1}$$

where $X_l \in \mathbf{R}^{[N_l]}$ and $X_{l,n} \in \mathbf{R}^{[N_l]}$ are zero–mean iid (independent identically distributed) random vectors representing the inputs and weights and N_l is the number of inputs of layer l. The variances of the components of these vectors are $\sigma_{x_l}^2$ and $\sigma_{w_l}^2$. Weight errors occur in all nodes of the network, causing input perturbations in the subsequent layers. Therefore, a stochastic model for the output of node n in layer l in a network with errors is given by:

$$Y_{l,n}^* = \tanh((X_l + \Delta X_l)^t(W_{l,n} + \Delta W_{l,n})) \tag{2.2}$$

where $\Delta X_l \in \mathbf{R}^{[N_l]}$ and $\Delta X_{l,n} \in \mathbf{R}^{[N_l]}$ are zero–mean iid random vectors representing the inputs and weight errors. The variances of the components in these vectors are $\sigma_{\Delta x_l}^2$ and $\sigma_{\Delta w_l}^2$. A stochastic model for the output error in node n of layer l is given by:

$$\Delta Y_{l,n} = \tanh((X_l + \Delta X_l)^t(W_{l,n} + \Delta W_{l,n})) - \tanh(X_l^t W_{l,n}) \tag{2.3}$$

The noise–to–signal ration (*NSR*) is defined as the ratio of variance of the output error in layer l to the variance of the output in layer l,

$$NSR_l = \frac{\sigma_{\Delta y_l}^2}{\sigma_{y_l}^2} \tag{2.4}$$

The *NSR* is very useful for understanding how errors propagate through the network. This is shown in the flow diagram of figure 2.1. Considering small weight errors (typically, the magnitude of weight errors is less than 5% of the weights) and a large number of nodes per layer (> 25) the Central Limit Theorem and the definition of variance give a recurrent definition of the output *NSR* of a multilayer perceptron [33], [34].

$$NSR_l = g_l(\sqrt{N_l}\sigma_x\sigma_{w_l})(NSR_{l-1} + \frac{\sigma^2_{\Delta w_l}}{\sigma^2_{w_l}}) \tag{2.5}$$

where

$$g_l(x) = \frac{x^2 \int_{-\infty}^{\infty} \frac{4e^{-\frac{s^2}{2}}}{\sqrt{2\pi}(1+\cosh(2xs))^2}\, ds}{\int_{-\infty}^{\infty} (\tanh(xs))^2 \frac{1}{\sqrt{2\pi}} e^{-\frac{s^2}{2}}\, ds} \tag{2.6}$$

is the *NSR* gain. This gain factor models the amplification of the *NSR* from layer to layer. A proof of (2.5) and (2.6) and models of non–sigmoidal neural networks are given in [33] and [34].

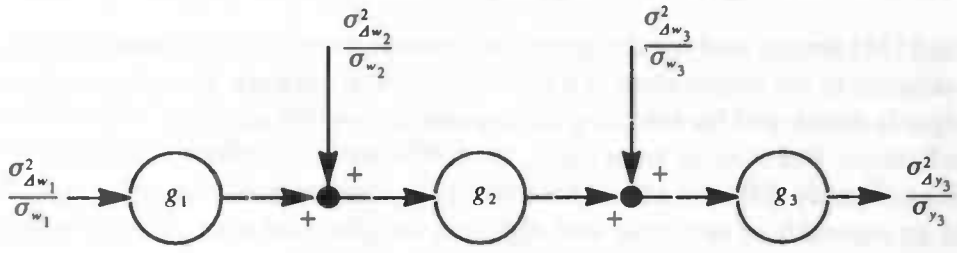


Figure: 2.1: *NSR signal flow diagram of a three layer sigmoidal neural network*

Example. This example illustrates an application of the noise–to–signal ratio approach. Suppose a designer is interested in building a hardware system with reduced precision weights that is capable of realizing three layer sigmoidal neuron networks with 256 nodes per layer (i.e. $L = 3$ and $N_l = 256$). The designer chooses a noise–to–signal ratio of $-20dB$ ($NSR = 0.01$) and bounds the weight of the hardware implementation to the interval $[-0.3, 0.3]$. The weights of the possible ensemble of ideal networks are assumed to be uniformly distributed in this interval (i.e. $\sigma_w^2 = 0.03$). Also, the designer limits the inputs of the hardware realization to the interval $[-1, 1]$. Given this squashing in the first layer and in the subsequent layers by the sigmoid functions, the designer estimates to input variance of each layer to be $\sigma_x^2 = 1.0$. Because the number of inputs, and the variances of the inputs and weights are identical in each layer, the *NSR* gain, g, of each layer is also identical. In this particular case, because the product $\sqrt{N}\sigma_x\sigma_w = 2.77$, the *NSR* gain (2.6) of each layer is $g = 2.0$. The accuracy of the weights can be found using:

$$NSR = (g^3 + g^2 + g)\frac{\sigma^2_{\Delta w}}{\sigma^2_w} \tag{2.7}$$

which is derived from figure 2.1. Given $g = 2.0$, $NSR = -20dB$, and assuming the weight accuracy of all three layers being identical, the required weight resolution is:

$$\frac{\sigma_{\Delta w}}{\sigma_w} = 0.027 \tag{2.8}$$

Given (2.8), the magnitude of the weight errors must be less than 2.7% of the magnitude of the weights. In a digital realization (2.6) corresponds a weight accuracy of 6 bits.

2.4 Finite Precision Analysis

Limited–precision hardware is prone to errors. In [32] the source of these errors and their statistical properties is demonstrated. Based on this theoretical analysis of finite precision errors, a re-

quired precision selection method is presented and proved in [32]. In this section we limit ourselves to the finite precision analysis.

Two common methods used for finite precision computation are *truncation* and *rounding*. The truncation operator simply chops the q lowest order bits off a number and leaves the new lowest order bit, in the 2^rth place, unchanged. The rounding operator has a similar behavior with the addition that if the q bit value chopped off is greater than or equal to 2^{r-1}, the resulting value is incremented by 2^r; otherwise, it remains unchanged. The error generated by truncation or rounding techniques maybe considered to be a discrete random variable. The following subsections present the mean and variance of the generated errors. The errors are assumed to be independent of each other and of all inputs and outputs.

Truncation

Truncation generates an error which is uniformly distributed in the range $[-2^r + 2^{r-q}, 0]$ with each of the 2^q possible errors values being of equal probability. Therefore, $p_i = 2^{-q}$ and $x_i = -i2^{r-q}$, for $i = 0$ to $2^q - 1$. Mean and variance are computed by:

$$\mu = \sum_{i=2^q+1}^{2^q-1} p_i i 2^{r-q} = -2^{r-2q} \sum_{i=0}^{2^q-1} i = -\frac{2^r - 2^{r-q}}{2} \tag{2.9}$$

$$\sigma^2 = \sum_{i=0}^{2^q-1} 2^{-q}(-i2^{r-q} + \frac{2^r - 2^{r-q}}{2})^2 = 2^{2r}\frac{1 - 2^{-2q}}{12} \tag{2.10}$$

Rounding

Rounding generates an error which is uniformly distributed in the range $[-2^{r-1}, 2^{r-1} - 2^{r-q}]$ with each of the 2^q possible errors values being of equal probability. Therefore, $p_i = 2^{-q}$ and $x_i = i2^{r-q}$, for $i = -2^{q-1}$ to $2^{q-1} - 1$. Mean and variance are computed by:

$$\mu = \sum_{i=-2^{q-1}}^{2^{q-1}-1} 2^{-q} i 2^{r-q} = 2^{r-2q} \sum_{i=-2^{q-1}}^{2^{q-1}-1} i = -2^{r-q-1} \tag{2.11}$$

$$\sigma^2 = \sum_{i=-2^{q-1}}^{2^{q-1}-1} 2^{-q}(i2^{r-q} + 2^{r-q-1})^2 = 2^{2r}\frac{1 - 2^{-2q}}{12} \tag{2.12}$$

For a finite precision computation of a nonlinear operation of multiple variables, several error sources exist. A neuron is basically a nonlinear device, computing $y = \phi(xw)$. The variables x and w, have errors ϵ_x and ϵ_w respectively, whose sources are prior limited–precision manipulations. Two more errors are introduced, the finite precision multiplication of x and w generates error $\epsilon_\bullet$, and the finite precision nonlinear operator ϕ generates the other error, ϵ_ϕ. Therefore, the resulting finite precision result y^* is equal to:

$$\begin{aligned} y^* &\equiv \phi((w + \epsilon_w)(x + \epsilon_x) + \epsilon_\bullet) + \epsilon_\phi \\ &= \phi(wx + w\epsilon_x + x\epsilon_w + \epsilon_w\epsilon_x + \epsilon_\bullet) + \epsilon_\theta \\ &\approx \phi(wx) + (w\epsilon_x + x\epsilon_w + \epsilon_\bullet)\phi'(wx) + \epsilon_\theta \end{aligned} \tag{2.13}$$

where it is assumed that the error product $\epsilon_w\epsilon_x$ is negligible, and a first–order Taylor series approximation is used. All errors, including the error $w\epsilon_x + x\epsilon_w$ resulting from the limited–precision multiplication, are further propagated through the nonlinear operator, resulting in the total finite precision error:

$$\epsilon_y = (w\epsilon_x + x\epsilon_w + \epsilon_\bullet)\phi'(wx) + \epsilon_\theta \tag{2.14}$$

This error will become the input finite precision error of variable y for following operations, as shown in figure 2.2.

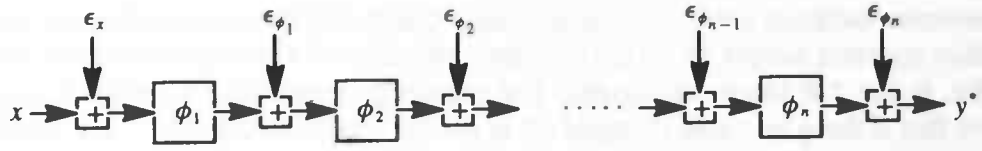


Figure 2.2: *Successive operators generate and propagate errors, where* $y = \phi_n(\ldots(\phi_2(\phi_1(x)))\ldots)$.

The error at the input, ϵ_x, and the error generated in each operator, ϕ, is propagated through the subsequent operators to the output. If y_i^* is defined as the intermediate result after the first i successive operators, then

$$\begin{aligned}
y_i^* &= \phi_i(\phi_{i-1}(\ldots((\phi_2(\phi_1(x+\epsilon_x)+\epsilon_{\phi_1})+\epsilon_{\phi_2})\ldots)+\epsilon_{\phi_{i-1}})\epsilon_{\phi_i} \\
&= \phi_i(y_{i-1}^*)+\epsilon_{\phi_i} \\
&= \phi_i(\phi_{i-1}(y_{i-2}^*)+\epsilon_{\phi_{i-1}})+\epsilon_{\phi_i} \\
&\approx \phi_i(\phi_{i-1}(y_{i-2}^*))+\phi_i'(\phi_{i-1}(y_{i-2}^*))\epsilon_{\phi_{i-1}}+\epsilon_{\phi_i} \\
&\approx \phi_i(\phi_{i-1}(\phi_{i-2}(y_{i-3}^*)))+\phi_i'(\phi_{i-1}(\phi_{i-2}(y_{i-3}^*)))\phi_{i-1}'(\phi_{i-2}(y_{i-3}^*))\epsilon_{\phi_{i-2}}+\phi_i'(\phi_{i-1}(y_{i-2}^*))\epsilon_{\phi_{i-1}}+\epsilon_{\phi_i} \\
&\approx \vdots \quad \vdots.
\end{aligned} \tag{2.15}$$

Carrying out similar expansion for all intermediate values, y^* can be rewritten:

$$y^* \equiv y_n^* \approx \phi_n(\phi_{n-1}(\ldots(\phi_2(\phi_1(x)))\ldots))+\sum_{i=0}^{n}\epsilon_{\phi_i}\prod_{k=i+1}^{n}\phi_k'(\phi_{k-1}(\ldots(\phi_{i+1}(y_i^*))\ldots)) \tag{2.16}$$

where $\prod_{k=n+1}^{n}\phi_k'(.)$ is defined to be 1. The product shown is just the chain rule for the derivative $\partial y/\partial y_i^*$, which can be further approximated by the derivative without error, $\partial y/\partial y_i$. This approximation is equivalent to the approximation already made in the first–order Taylor series.

$$\begin{aligned}
\frac{\partial y}{\partial y_i^*} &= \prod_{k=i+1}^{n}\phi_k'(\phi_{k-1}(\ldots(\phi_{i+1}(y_i^*))\ldots)) \\
&\approx \prod_{k=i+1}^{n}\phi_k'(\phi_{k-1}(\ldots(\phi_{i+1}(y_i))\ldots)) = \frac{\partial y}{\partial y_i}
\end{aligned} \tag{2.17}$$

Therefore,

$$y^* = y+\epsilon_y \approx y+\epsilon_x\frac{\partial y}{\partial x}+\sum_{i=1}^{n}\epsilon_{\phi_i}\frac{\partial y}{\partial y_i} \tag{2.18}$$

In case of multiple input operators, the effect of the finite precision error at the output of compound operators can be calculated through an extension of equation (2.18). This is accomplished by first breaking the computation into a *calculation graph*, which is illustrated in figure 2.3. The calculation graph shows the forward retrieving operation of a perceptron of a MLP.

By extending (2.18) to multiple inputs, and assuming n operators and m system inputs, the total finite precision error, ϵ_y, is given as:

$$\epsilon_y \approx \sum_{j=1}^{m}\epsilon_{x_j}\frac{\partial y}{\partial x_j}+\sum_{i=1}^{n}\epsilon_{\phi_i}\frac{\partial y}{\partial y_i} \tag{2.19}$$

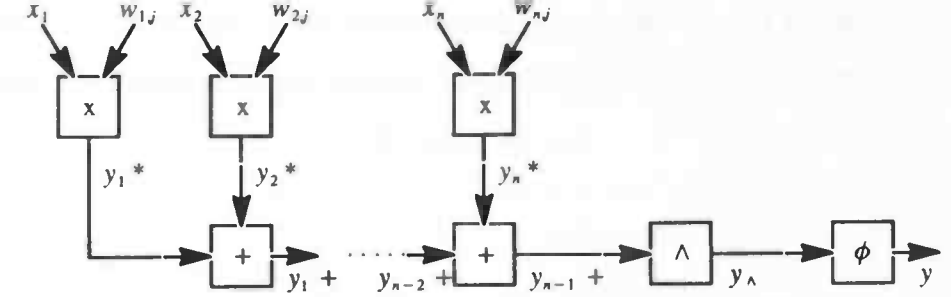


Figure 2.3: *Calculation graph of a neuron, where ∧ denotes a truncation or a rounding operator.*

The partial derivatives, $\partial y/\partial x_j$ and $\partial y/\partial y_i$ are computed from $y_j = \phi(\sum_i w_{ij}x_i)$, which constitutes a perceptron:

$$\frac{\partial y}{\partial x_i} = \phi_j' w_{ij} \quad \text{and} \quad \frac{\partial y}{\partial w_{ij}} = \phi_j' x_i \tag{2.20}$$

where $\phi_j' = \frac{1}{e^x(1 + e^{-x})^2}$, due to the nice property of the sigmoid function. In the calculation graph of figure 2.3, three additional errors are introduced: finite precision multiplication error, finite precision adding error and the truncation or rounding error. The partial derivatives of these errors are (see [32] for details):

$$\frac{\partial y}{\partial y_{i*}} = \phi_j' \quad \text{and} \quad \frac{\partial y}{\partial y_{i+}} = \phi_j' \quad \text{and} \quad \frac{\partial y}{\partial y_{i\wedge}} = \phi_j' \tag{2.21}$$

By substituting the partial derivatives in (2.19), gives:

$$\epsilon_y \equiv \phi_j' \sum_{i=1}^{n} w_{ij}\epsilon_\xi + \phi_j' \sum_{i=1}^{n} x_i \epsilon_{w_{ij}} + \phi_j' \sum_{i=1}^{n} \epsilon_{y_{i*}} + \phi_j' \sum_{i=1}^{n-1} \epsilon_{y_{i+}} + \phi_j' \epsilon_{y\wedge} + \epsilon_{\phi_j} \tag{2.22}$$

Equation (2.22) is a powerful tool for statistically evaluating the sensitivity of MLPs to limited precision arithmetic; by using the mean and variance of the truncation or rounding operator the mean and variance of the output error, ϵ_y, can be determined. The model can also be used to (roughly) estimate the required activation precision (e.g. the number of bits required for the sigmoid table–lookup) and the required weight precision for a hardware MLP realization. The reader is refered to [32] for a detailed example of the presented theory.

Chapter 3

Redundancy Reductions

3.1 Introduction

As noted earlier, a trained neural network can have a high degree of redundancy. We can measure and exploit this redundancy in transforming the trained ideal network into a network with approximately the same behavior which can be implemented at the expense of reduced fault tolerance with less costs. The introduction of lesser fault tolerance in the transformed network poses no threat, the failure of a connection or even a complete neuron is very unlikely considering a μ–processor based neural network implementation.

From the literature [1]–[3] we know that a trained neural network requires less precision in its weights. Further, a trained neural network does not use the complete domain of the activation functions. This effect is demonstrated by figure 3.1, which shows the actual activation function behavior of a singe input, singe output neural network with 6 hidden units (1–6–1 network) trained to simulate a halve period sine function: $0.1 + 0.8\sin(\frac{1}{2}\pi(x + 1))$, where $x \in [-1, 1]$.

In this chapter we describe several transformation (simplification) processes that try to optimize (simplify) particular trained neural network which can be implemented with less costs. We do this by determining weight accuracy and selecting alternative activation functions. The latter is not performed by the theory and methods presented in the previous chapter, they merely try to determine the required weight accuracy based on sensitivity analysis on an ensemble of neural networks.

The transformation process has to ensure that the sigmoidal and simplified neural network both have a similar behavior. This behavioral similarity is expressed by a user specification, ie an error measure. Further, the transformation process has to follow the four functional specification described in section 1.7, which we repeat here:

- The simplification process has to ensure that the ideal and the simplified neural network are both operating within a user specification, i.e. it has to be behavior–invariant.
- If possible, the sigmoid activation functions are replaced by simpler activation functions that are cheaper to evaluate.

- The required number of bits of each connection is determined and set accordingly.
- Irrelevant connections and neurons are determined and removed (pruned).

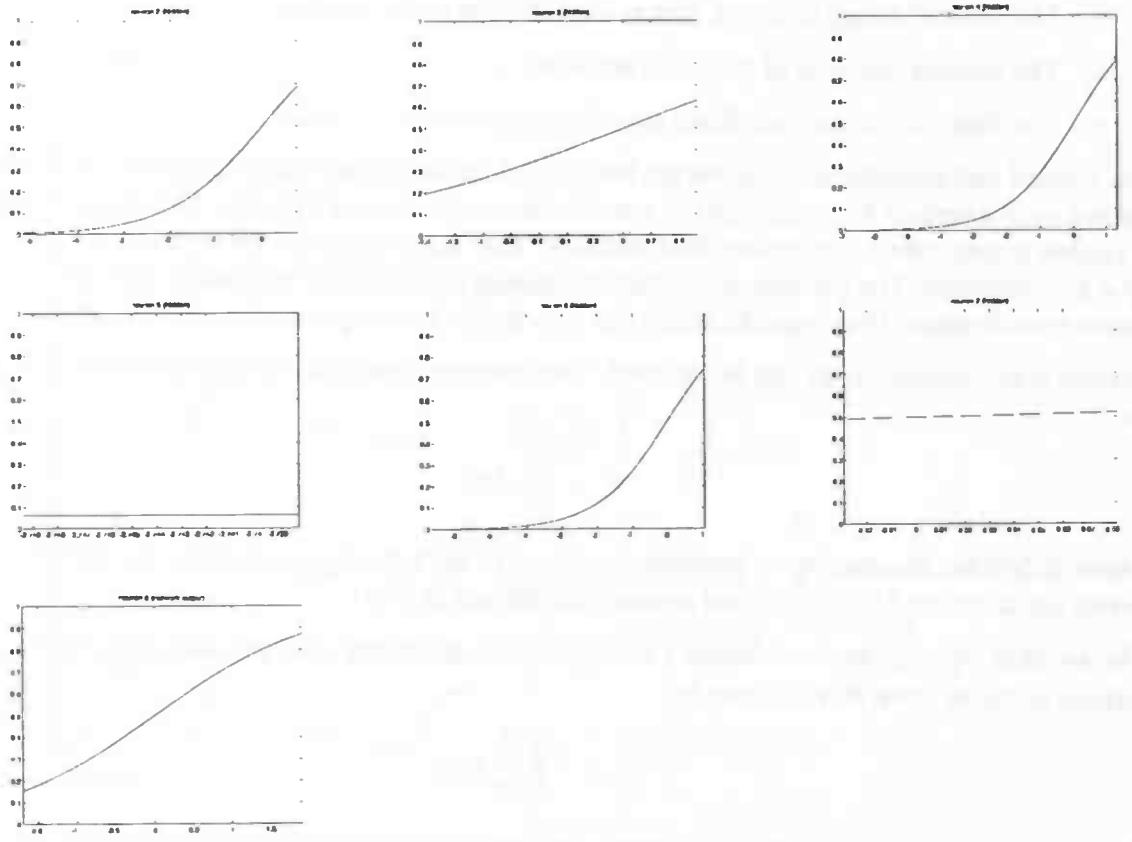


Figure 3.1: *The actual transfer behaviour of the trained "sinus"–network:* $0.1 + 0.8\sin(\frac{1}{2}\pi(x + 1))$. *The upper two rows show the plots for the hidden neurons, whereas the plot for the output neuron is shown in the bottom row. Notice that the input range (horizontal axis) differs for each plot. These plots show clearly that none of the neurons uses the complete sigmoid transfer function.*

3.2 Specifications

Specifications allow a designer to express the relation between the ideal network and its desired simplified version. The specifications enhance the network redundancy. A wider specification allows more redundancy to be removed.

The specifications are all based upon the allowable difference between the output of the simplified neural network and the target output, defined as:

$$e_j(n) = d_j(n) - t_j(n) \tag{3.1}$$

where $d_j(n)$ denotes the output j of the simplified network with pattern n at the input, $t_j(n)$ denotes the target output and $e_j(n)$ denotes the allowable error between the target and the simplified function. In this error expression there is no notation of the actual trained function. The trained function lies somewhere between the simplified result and the target function, as defined in (3.2):

$$|e_{trained}(n)| \leq |e_{simplified}(n)| \tag{3.2}$$

One has to consider three related neural networks or three related functions:

- The disered target function, this is the function to be trained.
- The trained function of the ideal network.
- The function of the simplified neural network.

The trained and simplified functions are both based on the desired target function. A network is trained until a certain stop criterium or specification (see chapter 2) holds. This trained network is further transformed into a simplfied network. This transformation process is also controlled by a specification. The training stop criterium should therefore be "narrower" than the simplification specification. Most specifications can also be used as stop criteria in a learning algorithm.

Several specification types can be defined. The instantaneous sum of squared errors is defined as:

$$\mathcal{E}_j = \sum_{n=1}^{N} e_j^2(n) \tag{3.3}$$

where N denotes the number of patterns contained in the test set, $e_j(n)$ denotes the difference between the target and the simplified network as defines in (3.1).

The average squared error of output j is obtained by summing $e_j^2(n)$ and then normalizing with respect to the set size N, as shown by:

$$\mathcal{E}_{AVGj} = \frac{1}{N}\sum_{n=1}^{N} e_j^2(n) \tag{3.4}$$

Another approach is to define a maximum absolute error between the target and the simplified network, as:

$$\mathcal{E}_{MAXj} = MAX_{n=1}^{N}|e_j(n)| \tag{3.5}$$

Sometimes it is desirable to define an error at a pattern by pattern basis, this allows more control in the simplification process. We define an error at a pattern by pattern basis for pattern n as follow:

$$\mathcal{E}_{pattern}(n) \tag{3.6}$$

The list of error functions presented above is not complete, it merely shows the most useful specifications. Other specification are left to the reader's imagination.

3.3 Activation Function Taxonomy

Several activation function exists, as shown in section 1.2, but training is performed with sigmoidal activation functions. This sigmoidal function (1.5) is very expensive to evaluate, it requires table lookup with interpolation or a Taylor series expension. On the other hand, a linear activation is far less expensive.

The evaluation cost of a neural network is greatly determined by the evaluation cost of the activation functions. Sigmoidal activation functions are nor always necessary during the evaluation phase (usage) of a neural network. The right choice of the activation function is therefore very important and is considered in section 3.4 and 3.5.

Activation function can be classified by evaluation cost as shown in the activation function taxonomy in figure 3.2. A cheaper function can introduce a larger error if a certain behavior of a activation function is needed is a domain not present in the cheaper activation function.

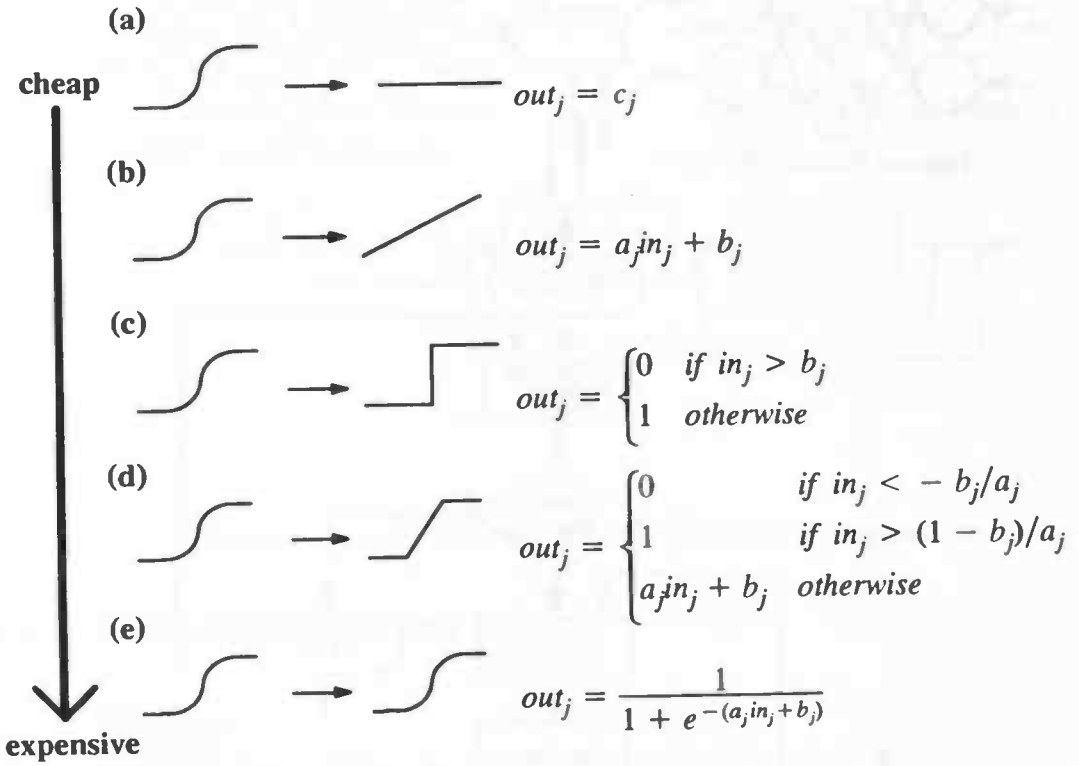


Figure 3.2: *Activation function taxonomy. A replacement with a constant function offers the cheapest replacement. A sigmoid to sigmoid replacement (no replacement) is the most expensive.*

3.4 Global Approach

The global approach is a very simple transformation algorithm. The algorithm starts by selecting a sigmoidal neuron at random and tries to replace the sigmoidal activation function with a function which is cheaper to evaluate. An activation function next in line in the activation taxonomy is chosen (i.e. a threshold function). After a replacement the network is tested to ensure the specifications are met. If this test succeeds, a next even more cheaper activation is tried. Otherwise the activation function is replaced by the previous chosen activation function. This simplification process ensures a behavior invariant transformation which is controllable with a user–specification and is shown schematically in figure 3.3.

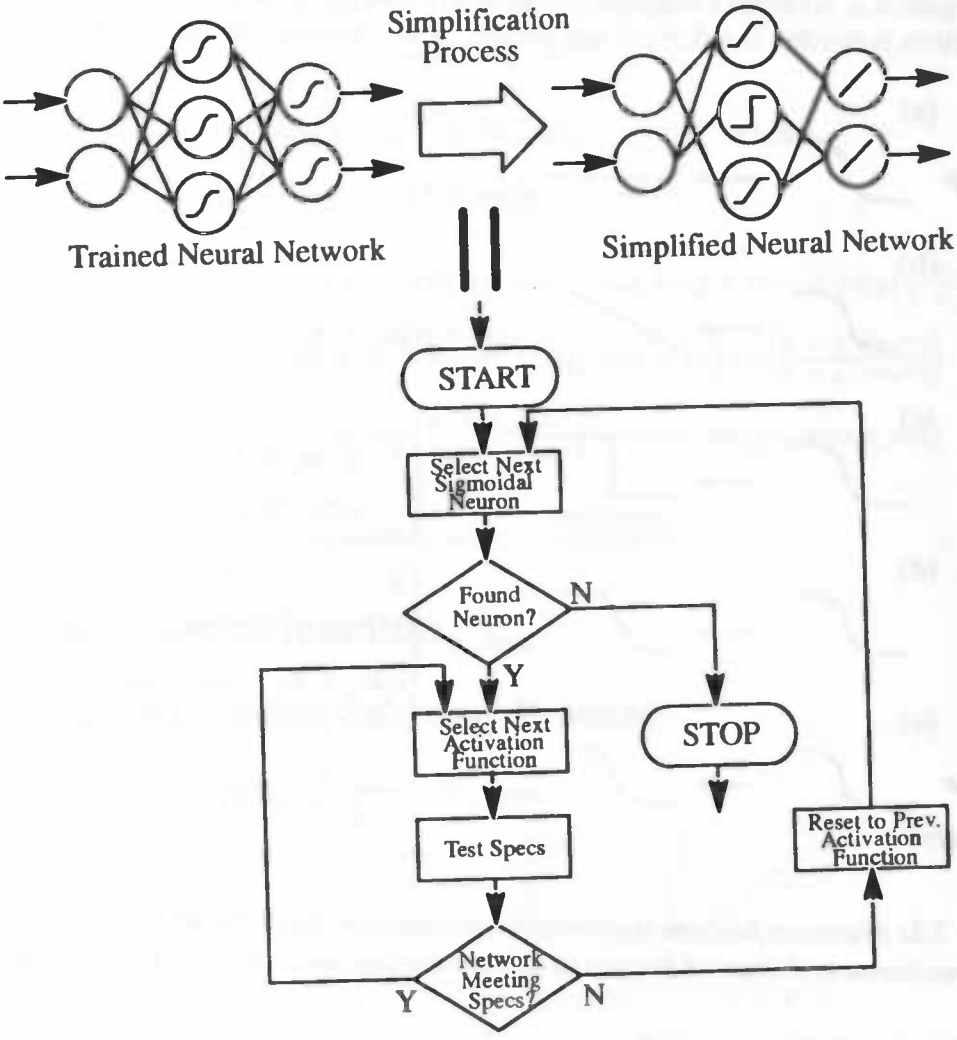


Figure 3.3: *Schematic view of the activation function replacement algorithm following the global approach.*

3.5 Local Approach

The global approach is

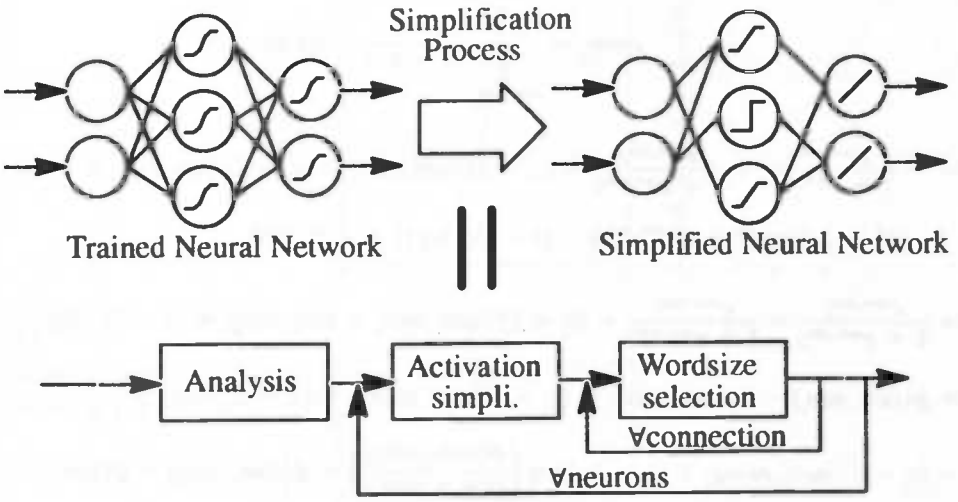


Figure 1.6: *Schematic view of the local simplification approach.*

3.5.1 Redundancy Indices

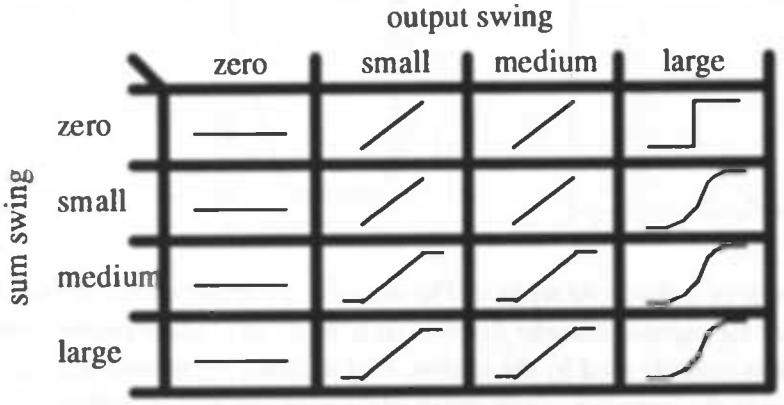


Figure 1.7: *Fuzzy activation function replacement decision table.*

$$sum\_min_j = \underset{\forall \text{ test patterns}}{MIN} \left(\sum_{i=1}^{N} w_{ij} out_i + \theta_j \right) \tag{1.13}$$

$$sum\_max_j = \underset{\forall \text{ test patterns}}{MAX} \left(\sum_{i=1}^{N} w_{ij} out_i + \theta_j \right) \tag{1.14}$$

$$sum\_swing_j = sum\_max_j - sum\_min_j \tag{1.15}$$

$$output\_swing_j = \phi(sum\_max_j) - \phi(sum\_min_j) \tag{1.16}$$

(1.17)

$$error_j = \int_{sum\_min_j}^{sum\_max_j} (\frac{1}{1+e^{-x}} - c_j)^2 dx$$

$$error_j = -\frac{e^{sum\_max_j}}{1+e^{sum\_max_j}} + \frac{e^{sum\_min_j}}{1+e^{sum\_min_j}} + (c_j - 1)^2 sum\_max_j - (c_j - 1)^2 sum\_min_j +$$
$$(1 - 2c)\log(1 + e^{-sum\_max_j}) - (1 - 2c)\log(1 + e^{-sum\_min_j})$$
$$= \frac{e^{sum\_min_j}}{1+e^{sum\_min_j}} - \frac{e^{sum\_max_j}}{1+e^{sum\_max_j}} + (c_j - 1)^2(sum\_max_j - sum\_min_j) + (1 - 2c_j)\log\left(\frac{1+e^{-sum\_max_j}}{1+e^{-sum\_min_j}}\right)$$
$$= \phi(sum\_min_j) - \phi(sum\_max_j) + (c_j - 1)^2 sum\_swing_j + (1 - 2c_j)\log\left(\frac{1+e^{-sum\_max_j}}{1+e^{-sum\_min_j}}\right)$$
$$= (c_j - 1)^2 sum\_swing_j + (1 - 2c_j)\log\left(\frac{\phi(sum\_min_j)}{\phi(sum\_max_j)}\right) + \phi(sum\_max_j) - \phi(sum\_min_j) \quad (1.18)$$

$$redundancy\_index_j = \frac{sum\_swing_j}{error_j} \quad (1.19)$$

3.6 Sigmoid Simplificaction

3.6.1 Activation Function Selection Process

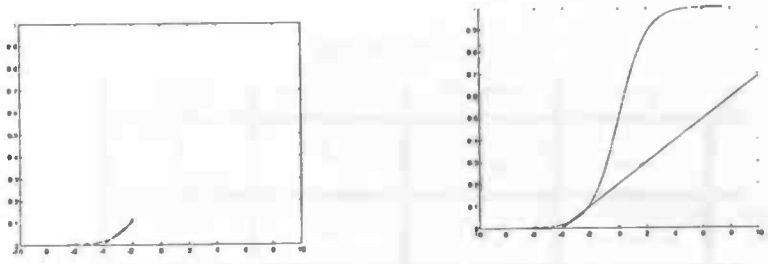
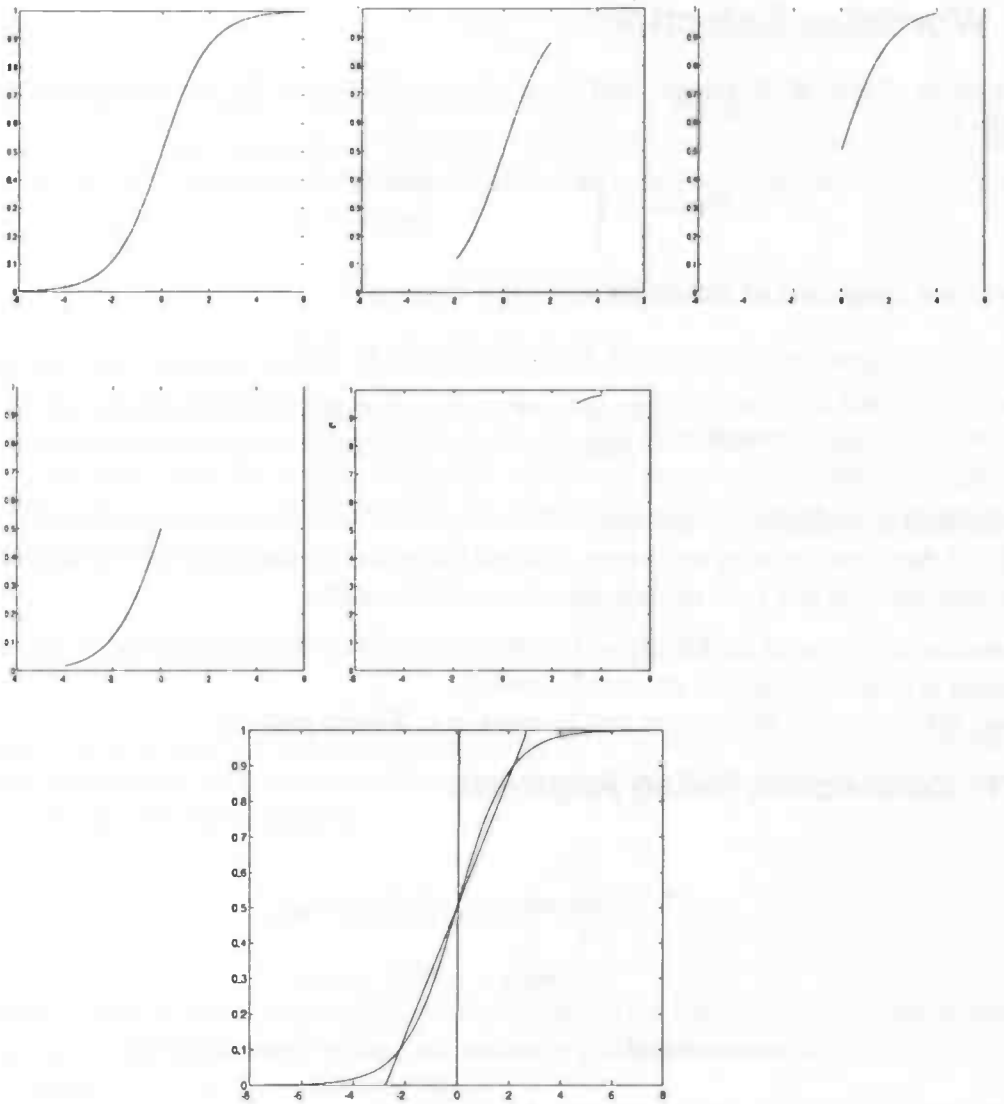


Figure 1.8: *An example of a threshold replace. The actual slope of the threshold function can differ from the initial slope of the sigmoid transfer function as a"best" fit is made only with the part of the sigmoid function that is actually used by the neuron. Plot A shows the neuron response in the used area only. Plot B covers the complete output range of the sigmoid. A large difference between the threshold and the sigmoid occurs only for unused input sums.*



met een fout afschatting

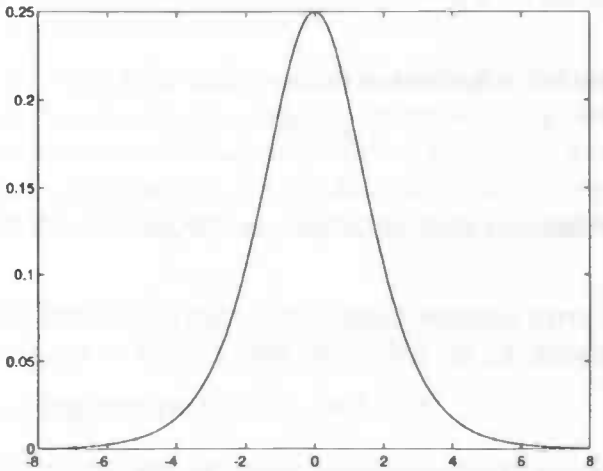


Figure 1.9: *derivative .....*

3.7 Wordsize Selection

The number of bits of the integer part is easialy determined by the following formula (add 1 for the sign bit).

$$\#intbits = \begin{cases} 1 + \lceil {}^2\log(|weight|) \rceil & , |weight| > 0 \\ 1 & , |weight| \leq 0 \end{cases} \tag{1.20}$$

Weights are composed of an integer and a fractional part.

$$0 \leq fractional\ part \leq 1\text{–}2^{n} \tag{1.21}$$

$$weight = \begin{cases} integer\ part + fractional\ part & , weight \geq 0 \\ integer\ part - fractional\ part & , weight < 0 \end{cases} \tag{1.22}$$

The problem is to determine the required number of bits of the fractional part of a weight. One could call this a bit–pruning technique. Several statistical methods exists. The algorithms are but a few exceptionals not very usefull, nice theoretical results.

Information is to coarse to determine the exact number of bits required by the fractional part. Algoritm is controlled by the connection swing.
Several other pruning techniques can be used (e.g. Karnin prunig).

3.7.1 Connection Swing Approach

$$sum\_min_{ji} = \underset{\forall test\ patterns}{MIN}\ w_{ji} out_i$$

$$sum\_max_{ji} = \underset{\forall test\ patterns}{MAX}\ w_{ji} out_i$$

$$connection\_swing_{ji} = connection\_max_{ji} - connection\_min_{ji} \tag{1.23}$$

3.7.2 Connection Sensitivity Approach

Upon completion of training we are equipped with a list of sensitivity numbers, one per each connection. They are created by a process that runs concurrently, but without interfering, with the learning process.

The global error of the net is defined as an extension of (1.9):

$$E = \sum_{n} \sum_{j} e_j(n)^2 \tag{1.24}$$

where the inner summation is over all output neurons, and the outer sum is over the patterns of the training set.

The sensitivity of the error function to the elimination of a connection can be estimated. In terms of connection elemination S_{ji}, the sensitivity with respect to w_{ji} is defined as:

$$S_{ji} = E(w_{ji} = 0) - E(w_{ji} = w_{ji}^f) \tag{1.25}$$

where w_{ji}^f is the final value of the connection upon the completion of the training phase. Equation (3.) can be rewritten as:

$$S_{ji} = -\frac{E(w^f) - E(0)}{w^f - 0} w^f \tag{1.26}$$

where $w = w_{ji}$ and E is expressed as a function of w, assuming that all other weights are fixed (i.e. at their final states, upon completion of learning). A typical learning process doen not start with $w_{ji} = 0$, but rather with some small random choses initial value w_{ji}^i. Since we do not known $E(0)$, we will approximate S_{ji} by:

$$S_{ji} \approx -\frac{E(w^f) - E(w^i)}{w^f - w^i} w^f \tag{1.27}$$

The initial and final weights, w_{ji}^i and w_{ji}^f, respectively, are quantities that are readily available during the training phase. However, here it is assumed that only one weight, namely w_{ji} had been changed, while all others remained fixed. This is clearly not the case during normal training. Consider a network with only two weights, denoted u and w (the extension to more weight will become obvious). In case we are interested in the contribution (S) of weight w, the numerator of (3.) can be evaluated as:

$$E(w = w^f) - E(w = 0) = \int_A^F \frac{\partial E(u^f, w)}{\partial w} dw \tag{1.28}$$

The integral starts in point A, which corresponds to $w_{ji} = 0$ (while all other weights are in their final states), to the final weight state F. However, the training phase starts at point I, so we have to approximate the above integral to:

$$E(w = w^f) - E(w = 0) \approx \int_I^F \frac{\partial E(u, w)}{\partial w} dw \tag{1.29}$$

This expression will be further approximated by replacing the integral by summation, taken over the discrete steps that the network passes while learning. Thus the estimated sensitivity to the removal of connection w_{ji} will be evaluated as:

$$S_{ji}^* = -\sum_{n=0}^{N-1} \frac{\partial E}{\partial w_{ji}} \Delta w_{ji}(n) \frac{w_{ji}^f}{w_{ji}^f - w_{ji}^i} \tag{1.30}$$

where N is the number of training cycles (epochs).

The above estimate for the sensitivity uses terms that are readily available during training. Obviously the weight increments Δw_{ji} are the essence of every learning process, so they are always available. Also, most optimization search uses gradients to find the direction of change
For the special case of back–propagation, weights are updated according to (1.10), hence (3.) reduces to (for learing with momentum, the general form of (3.) should be used):

$$S_{ji}^* = \sum_{n=0}^{N-1} [\Delta w_{ji}(n)]^2 \frac{w_{ji}^f}{\eta(w_{ji}^f - w_{ji}^i)} \tag{1.31}$$

3.8 Linear Regression

PLAATJE!! In order to determine the angle of the threshold function

$$f(a,b) = \sum(Y_i - a - bX_i) = \sum e_i^2 \tag{1.32}$$

$$\frac{\partial}{\partial a}f(a,b) = 0 \quad \text{and} \quad \frac{\partial}{\partial b}f(a,b) = 0 \tag{1.33}$$

Function f is a quadratic function with positive terms, so the extreme value of ... is a minimum.

$$\frac{\partial}{\partial a}\sum e_i^2 = \frac{\partial}{\partial a}\sum(Y_i - a - bX_i)^2 = -2\sum(Y_i - a - bX_i) \tag{1.34}$$

$$\frac{\partial}{\partial b}\sum e_i^2 = \frac{\partial}{\partial b}\sum(Y_i - a - bX_i)^2 = -2\sum X_i(Y_i - a - bX_i)$$

$$\sum Y_i = na + b\sum X_i \tag{1.35}$$

$$\sum X_i Y_i = a\sum X_i + b\sum X_i^2 \tag{1.36}$$

$$b = \frac{n\sum X_i Y_i - \sum X_i \sum Y_i}{n\sum X_i^2 - (\sum X_i)^2} \tag{1.37}$$

$$a = \frac{\sum Y_i}{n} - b\frac{\sum X_i}{n} \tag{1.38}$$

The bias needs updating as well.

$$sum_j = \sum_{i=1}^{N} w_{ji} out_i + \theta_j \tag{1.39}$$

$$out_j = b_j sum_j + a_j \tag{1.40}$$

$$= b\sum_{i=0}^{N} w_{ij} out_i + b\theta_j + a \tag{1.41}$$

$$= b(\sum_{i=1}^{N} w_{ij} out_i + \theta_j + \frac{a}{b}) \tag{1.42}$$

$$= b(\sum_{i=1}^{N} w_{ij} out_i + \theta_j') \tag{1.43}$$

$$\theta_j' = \theta_j + \frac{a}{b} = \theta_j + \frac{\sum Y_i}{b_j n} - \frac{\sum X_i}{n} \tag{1.44}$$

The new bias.

Chapter 4

Experiments

4.1 Introduction

This chapter demonstrates the methods and theory presented in the previous chapter. A simple single input single output neural network is trained to simulate a halve period sinus function. Secondly, a more elaborate real engineering application is used to show the effect of the local approach. Finally, the speedup accomplished by the simplification process is demonstrated by a large neural network trained for performing a character recognition task. All simulations are carried out by the **InterAct** neural network tooling environment.

4.2 Sinus

The "sinus"–network is a simple sigmoidal single input single output neural network with six hidden neurons (1–6–1 network) trained to simulate a halve period sine function: $0.1 + 0.8\sin(\frac{1}{2}\pi(x + 1))$, where $x \in [-1, 1]$. This network may not seem to be very useful at a first glance, but the network size offers the possibility to show all of the network interior, which will help in better understanding the problem. Figure 4.1 shows the constructed network and the sine function to be simulated.

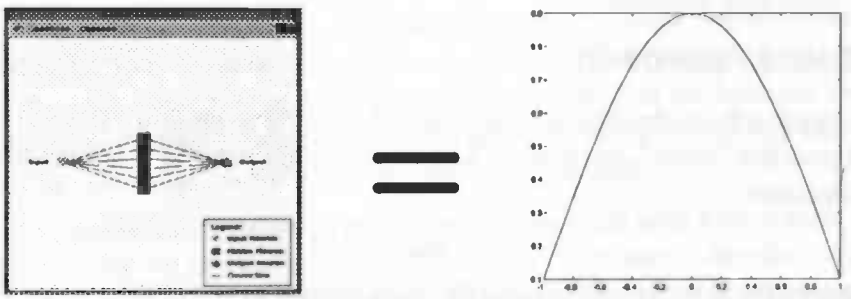


Figure 4.1: *"Sinus"–network set up.*

Training is performed until the mean squared error (MSE) is less than 0.001. The training and test set are both composed of 50 samples. Figure 3.2 shows the activation function behavior of the trained "sinus"–network. This figure clearly shows that the complete domain of the sigmoid activation function is not required. Subsequently, the network has a lot of redundancy in the activation functions, which will be removed.

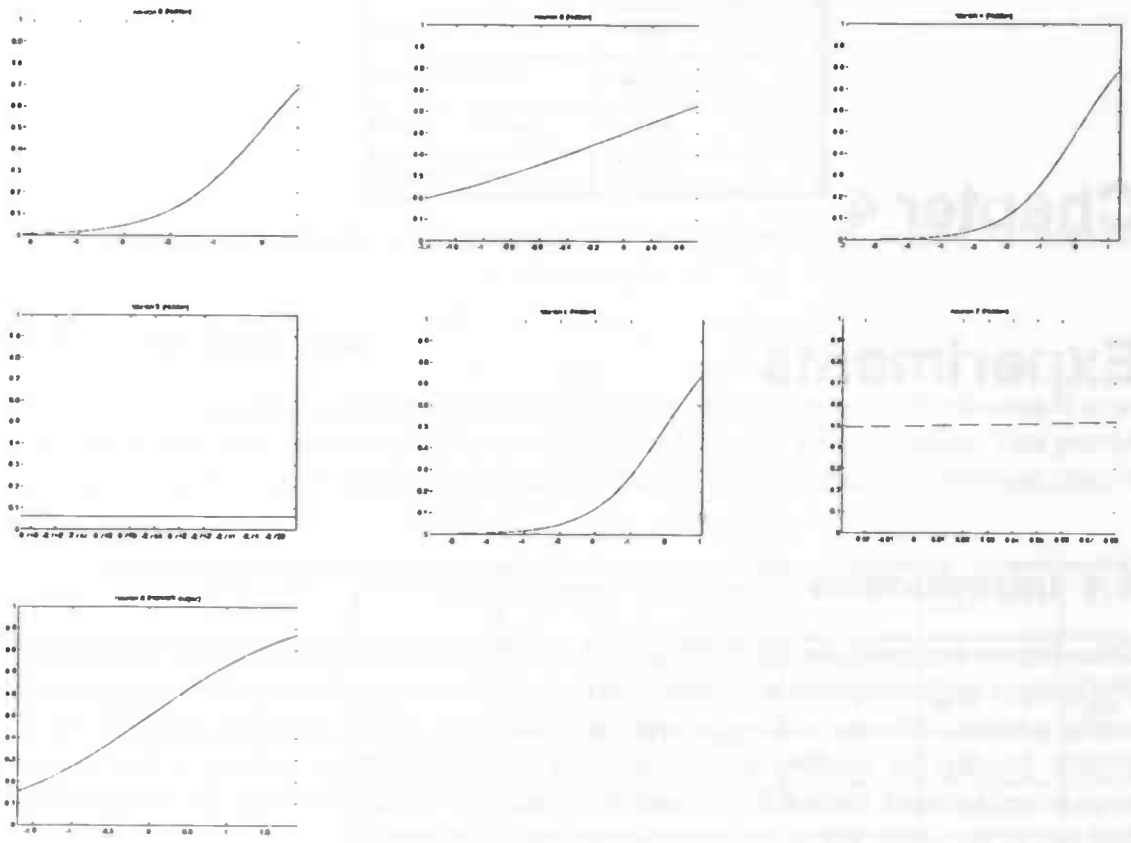


Figure 4.2: *The actual transfer behaviour of the trained "sinus"–network:* $0.1 + 0.8\sin(\frac{1}{2}\pi(x + 1))$. *The upper two rows show the plots for the hidden neurons, whereas the plot for the output neuron is shown in the bottom row. Notice that the input range (horizontal axis) differs for each plot. These plots show clearly that none of the neurons uses the complete sigmoid transfer function.*

The next two subsections demonstrate the global and local neural network simplification procedures, as proposed in the previous chapter. The global approach only addresses activation function simplifications, the local approach on the other hand, addresses activation function simplification and word size selection.

4.2.1 Global Approach

The global approach, as described in section 3.3, is used to simplify the activation functions of a trained sigmoidal "sinus"–network. The simplification process is controlled by the following user–specification:

$$\mathcal{E}_{AVG} = 0.05 \tag{1.45}$$

Table 4.1 shows the resulting activation function replacement (optimization phase requires well over 150 seconds as opposed to the 20 seconds training time!).

Neuron	New status
neuron 2 (hidden)	threshold
neuron 3 (hidden)	linear
neuron 4 (hidden)	threshold
neuron 5 (hidden)	removed
neuron 6 (hidden)	threshold
neuron 7 (hidden)	removed
neuron 8 (output)	linear

Table 4.1: *Activation function replacement using the global approach controlled by specification:* $\mathcal{E}_{AVG} = 0.05$.

We can repeat this experiment using a "wider" user–specification:

$$\mathcal{E}_{AVG} = 0.1 \tag{1.46}$$

This "wider" specification enhances the redundancy present in the trained neural network. The resulting activation function replacement is shown in table 4.2.

Neuron	New status
neuron 2 (hidden)	hardlimiter
neuron 3 (hidden)	linear
neuron 4 (hidden)	threshold
neuron 5 (hidden)	removed
neuron 6 (hidden)	threshold
neuron 7 (hidden)	removed
neuron 8 (output)	linear

Table 4.2: *Activation function replacement using the global approach controlled by specification:* $\mathcal{E}_{AVG} = 0.1$.

Hidden neuron 1 has been changed into a neuron equiped with a hardlimiter activation function. This results in a network that is a bit cheaper to evaluate than the preceding network. We may even wider the user–specification as follows:

$$\mathcal{E}_{AVG} = 0.2 \tag{1.47}$$

Given this wide user–specification, we expect a network that is even more cheaper to evaluate than the preceding ones. Looking at the resulting activation function replacement shown in table 4.3, we concludes that this is not the case. Neurons are simplified in order of their neuron number, ie neuron 2 is optimized first and neuron 8 last. The neurons that are selected first are replaced by cheap neurons that introduce large errors. This prevents neurons selected later from being replaced by cheaper ones without violating the user–specification.

We may conclude that using the global simplification approach with a very wide user–specification may result in a solution that is not necessarily cheaper to evaluate despite its enhanced redundancy. The global approach is too optimistic at the start of the simplification process, resulting in a network that is not necessarily cheaper to evaluate despite its enhanced redundancy.

Neuron	New status
neuron 2 (hidden)	hardlimiter
neuron 3 (hidden)	linear
neuron 4 (hidden)	hardlimiter
neuron 5 (hidden)	sigmoid
neuron 6 (hidden)	sigmoid
neuron 7 (hidden)	sigmoid
neuron 8 (output)	sigmoid

Table 4.3: *Activation function replacement using the global approach controlled by specification:* $\varepsilon_{AVG} = 0.2$.

4.2.2 Local Approach

In this section we use the local approach to simplify the "sinus"–network. As discussed in section 3.5 the local approach can not easily be parameterized by user specifications. This prevents us from controlling the transformation process. We can not assure a behavior invariant transformation.

Neuron	Min. input	Max. input	Min. ouput	Max. output	Avg. output	Redundancy index	New status
neuron 2 (hidden)	-6.91	0.80	0.00	0.69	0.15	25.40	threshold
neuron 3 (hidden)	-1.45	0.53	0.19	0.63	0.39	58.50	linear
neuron 4 (hidden)	-6.92	1.32	0.00	0.79	0.19	17.39	threshold
neuron 5 (hidden)	-4.56	-2.75	0.06	0.06	0.03	5.05e3	removed
neuron 6 (hidden)	-6.91	1.10	0.00	0.75	0.17	20.17	threshold
neuron 7 (hidden)	-0.04	0.08	0.49	0.52	0.51	1.43e4	removed
neuron 8 (output)	-1.73	1.90	0.15	0.87	0.51	19.39	linear

Table 4.4: *The range and redundancy values for the various neurons in the "sinus"–network.*

After simplification the network follows the following specifications: $\varepsilon_{AVG} = 0.05$ and $\varepsilon_{MAX} = 0.13$. We can clearly see that the output neuron's sigmoid activation function is replaced by a linear activation function.

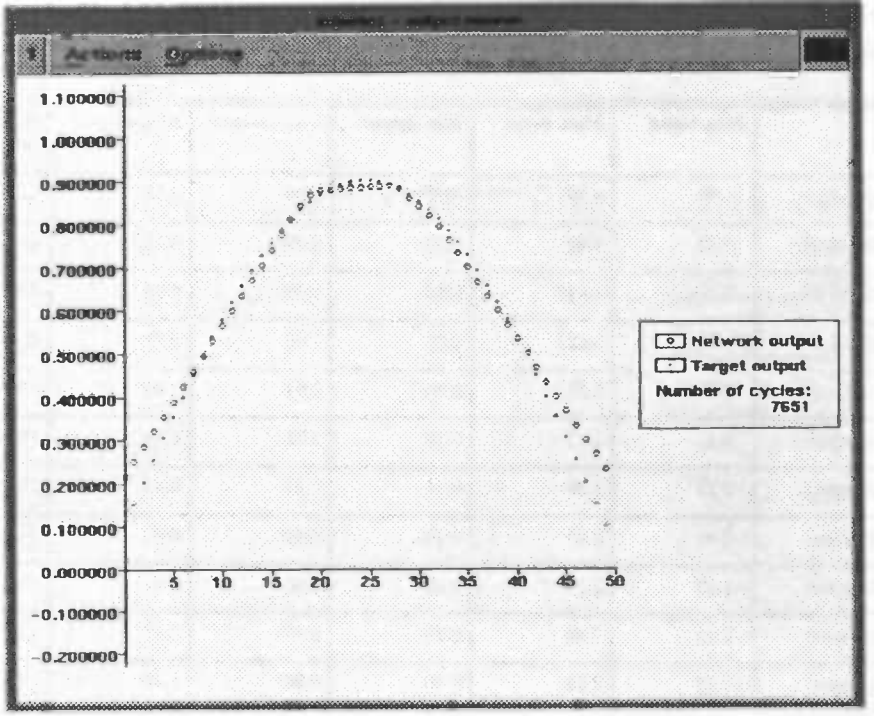


Figure 4.3: *Output of the simplified "sine"–neural network.*

4.3 Sand Grain Size–Distribution

The network used in the previous section is small and is not a very serious application. In this section a more elaborate real engineering neural network application is studied. In this application an advanced vision system is developed [36] for the dregging industry that can be used to determine size–distributions of sand particles in water. A 13–input, 4–hidden and 9–output sigmoidal feedforward neural network is constructed and trained with run length statistics of sand images to determine the corresponding size–distribution (D10..D90 network outputs), as shown in figure 4.4.

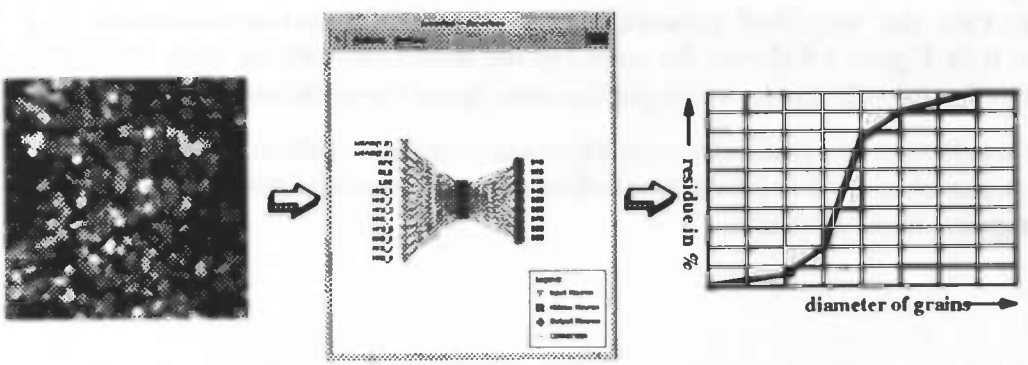


Figure 4.4: *System setup showing the input image, the structure of the used neural network and the resulting cumulative size–distribution of the sand sample*

After training the redundancy in the activation functions is removed by the local simplification approach. Connections are pruned using the connection swing method, they are removed if the

recorded swing is less than 0.05. Table 4.5 shows the range and redundacy values for the various neurons in the network, 2 connections could be pruned.

Neuron	Min. input	Max. input	Min. ouput	Max. output	Avg. output	Redundancy index	New status
neuron 14 (hidden)	-4.56	6.59	0.01	0.99	0.57	6.51	threshold
neuron 15 (hidden)	-5.92	4.45	0.00	0.99	0.16	4.44	sigmoid
neuron 16 (hidden)	-3.41	13.43	0.03	1.00	0.94	7.97	threshold
neuron 17 (hidden)	-17.53	30.21	0.00	1.00	0.78	4.27	sigmoid
neuron 18 (output)	-2.19	2.28	0.10	0.91	0.45	13.77	linear
neuron 19 (output)	-2.19	2.27	0.10	0.91	0.44	13.60	linear
neuron 20 (output)	-2.17	2.26	0.11	0.91	0.43	13.42	linear
neuron 21 (output)	-2.16	2.25	0.10	0.90	0.42	13.24	linear
neuron 22 (output)	-2.15	2.24	0.10	0.90	0.41	13.01	linear
neuron 23 (output)	-2.15	2.24	0.10	0.90	0.41	12.79	linear
neuron 24 (output)	-2.17	2.21	0.10	0.90	0.40	12.65	linear
neuron 25 (output)	-2.19	2.18	0.10	0.90	0.39	12.48	linear
neuron 26 (output)	-2.23	2.16	0.10	0.90	0.37	12.20	linear

Table 4.5: *The range and redundancy values for the various neurons in the "sand"–network.*

Looking at table 4.5, we notice that the output neurons are always converted to neurons with a linear activation function. This is because the output neurons are scaled to be within [0.1 0.9], so they operate solely in the linear region of the activation function. Despite their low redundancy indices, two hidden neurons could be converted to neurons with a threshold activation function. The other two hidden neurons remain unchanged. This shows that the selection of the sigmoid simplification switch–points are critical and have to be user–specification controlled. Unfortunately, the local approach can not easily be parameterized by such a user–specification. The local approach can be considered as merely a trial–and–error way of optimizing a neural network.

In this case the simplified network operates within the user–specifications $\varepsilon_{AVG} = 0.02$ and $\varepsilon_{MAX} = 0.28$. Figure 4.5 shown the output of the neural network for class H016 sand, the above figure is the sigmoidal network output the other figure shows the output of the simplified version.

If we would map this particular network to a µ–processor without a floating point unit, we may need to use table–lookup for the two hidden neurons that could not be simplified. This can make the mapping more difficult.

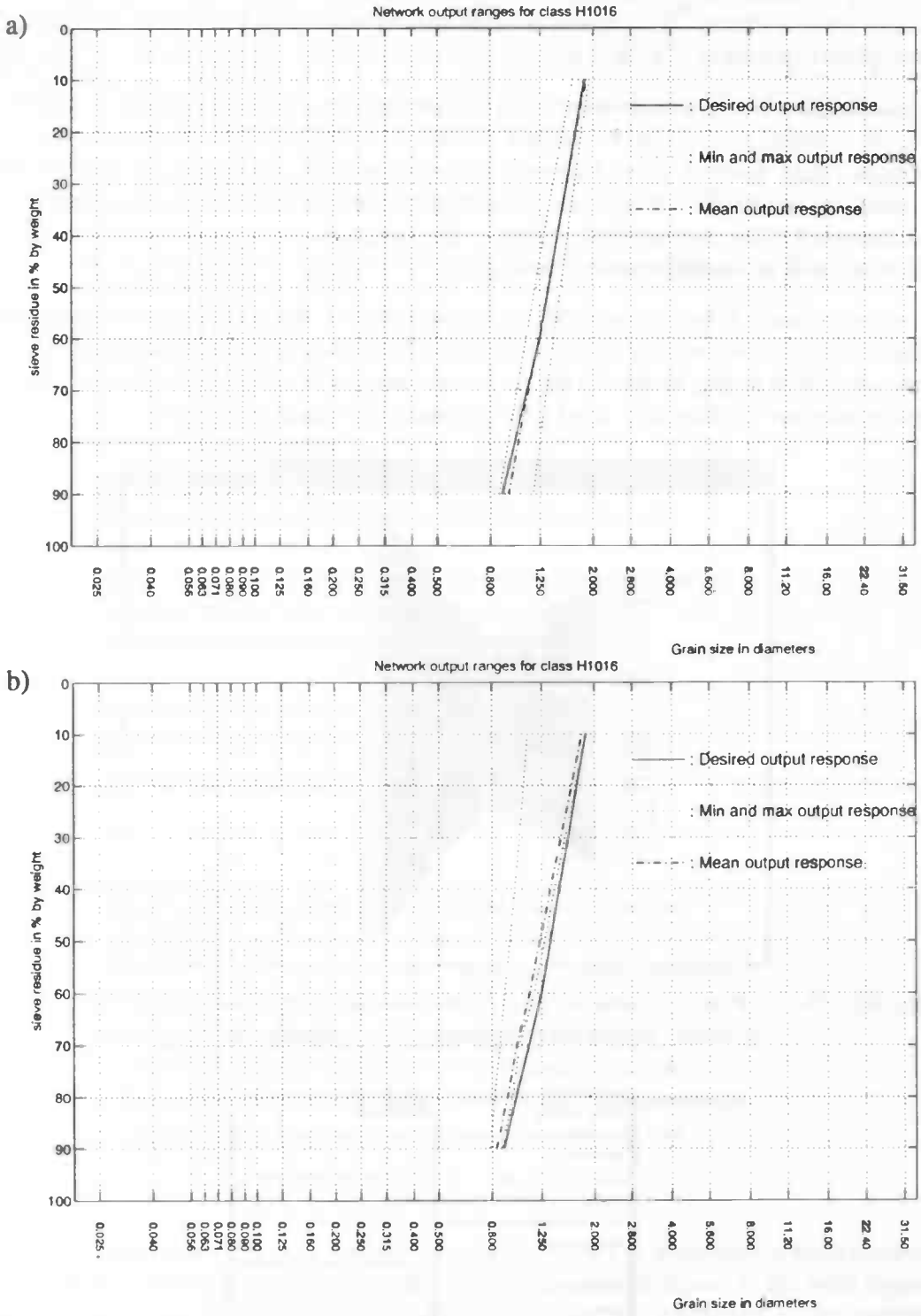


Figure 4.5: *logplot vertical axis respresent the D10..D90 size-distribution output of the neural network, the horiontal axis represents the grain size in diameter, a. = sigmoidal b = simplified.*

4.4 Optical Character Recognition

In this section we study the speedup achieved by the activation function simplification process following the local approach and the connection swing pruning algorithm. A large 24 input–, 15 hidden–and 36 output sigmoidal neural network is constructed and trained to perform a character regonition task, as shown in figure 4.6 . The network is used as a character recognizer in a car lincense plate regonition (CLPR–) system [37] developed at our laboratory.

The classification is performed on the basis of four features extracted from the image: horizontal projection, vertical projection, horizontal connected component count and vertical connected component count. Each of these features is transformed into six inputs for a MLP–neural network, resulting in a total of 24 inputs. A character can be A–Z0–9 (hence, 36 outputs), and is said to be recognized iff the output of the corresponding output neuron exceeds 0.85 and all other output neurons have an output level below 0.25.

Initially the network is trained with 51 sigmoidal neurons. After training the redundancy in the activation functions is removed by the local simplification approach. Connections are pruned using the connection swing method if the recorded swing is less than 0.05. The activation simplifiaction results are displayed in tabel 4.6, 9 connections could be pruned.

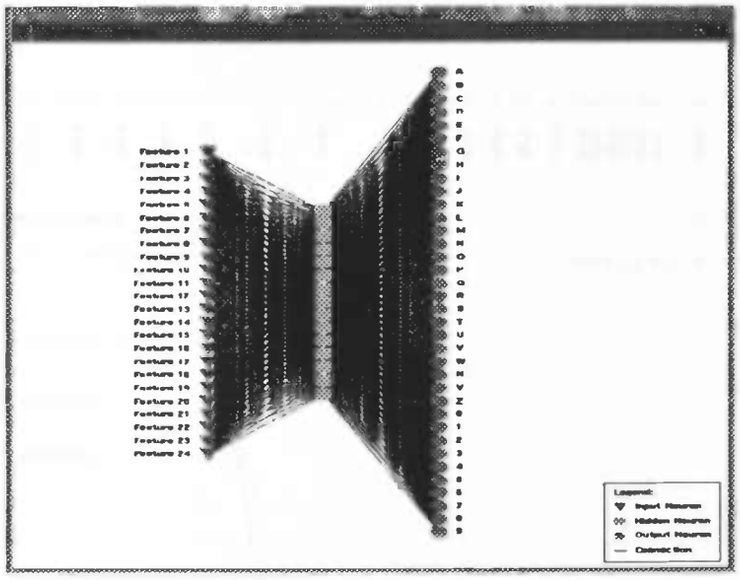


Figure 4.6: *The MLP–neural network used for the classification of characters. Network contains 24 input–, 15 hidden–and 36 output neurons.*

New status	#Neurons
removed	0
linear	30
hardlimiter	0
threshold	12
simgoid	9

Table 4.6: *New activation function types for the "OCR"–network.*

To be able to measure the achieved speedup, the simplified network and the original sigmoidal network have both been transformed into C–code. This process is illustrated in figure 4.7. After connection pruning a code generation module has been added.

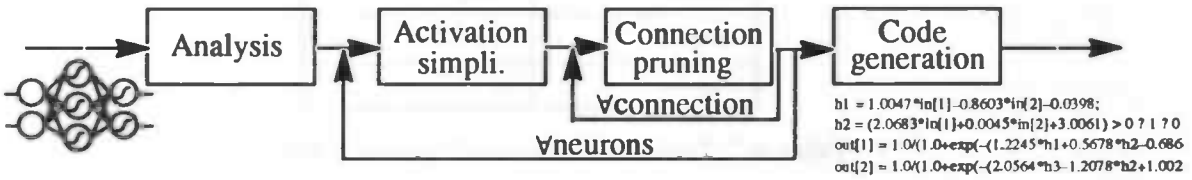


Figure 4.7: *Simplification and code generation process.*

The next code fragment shows a part of the generated C–code of the simplified network.

```
#define NR_INPUTS 24
#define NR_OUTPUTS 36

void InterActNetO(float *in, float *out)
{
  /*Variable declarations for the hidden neurons*/
  float h25,h26,h27,h28,h29,h30,h31,h32,h33,h34,h35,h36,h37,h38,h39;

  h25 = ((h25 = –0.571630*in[0]–0.209781*in[1]+1.428952*in[2]–0.316204*in[3]
    –2.259562*in[4]–0.671550*in[5]+0.704688*in[6]+0.197393*in[7]
    +0.844528*in[8]+0.841312*in[9]–0.189561*in[10]–2.537952*in[11]
    +0.235859*in[12]–0.563117*in[13]–1.602229*in[14]–0.412747*in[15]
    +3.482257*in[16]+2.151931*in[17]–0.494821*in[18]–0.760074*in[19]
    +1.235106*in[20]+1.361977*in[21]–0.749663*in[22]+1.720739*in[23]
    +0.811305) > 0.0 ? (h25 < 1.0 ? h25 : 1.0) : 0.0);
  .
  .
  .
  out[34] = 1.0/(1.0+exp( –(–0.850665*h25–1.054678*h26–0.977259*h27
    –0.837096*h28–0.918724*h29–1.059094*h30–0.954263*h31–1.060187*h32
    –0.867417*h33–0.683362*h34–0.784591*h35–0.669613*h36–1.159230*h37
    –0.988633*h38–0.715659*h39–2.176036) ));
  out[35] = ((out[35] = +0.000000*h25+0.000000*h26+0.000000*h27+0.000000*h28
    +0.728621*h29–1.057011*h30+0.597944*h31+0.708962*h32–0.816435*h33
    +0.740386*h34+0.563300*h35–0.596560*h36–0.065871*h37–0.586360*h38
    –1.486406*h39+0.020043) > 0.0 ? (out[35] < 1.0 ? out[35] : 1.0) : 0.0);
}
```

Table 4.7 shows the timing results performed on a HP9000/735 machine for both networks. The actual speedup achieved is machine depend. The timing results shown in the next figure can be considered worst case for the simplified network, because this particular target machine has a very fast floating point unit (the timing results are in fact useless). A much lager speedup is achieved when porting both network to e.g. a 8–bit machine without floating–point hardware.

The timing results show a large speedup of nearly 20%, the network still operates within the specifications set above.

Network	Timing results (msec)
sigmoidal	780
simplified	580

Table 4.7: *Timing results on a HP9000/735 machine.*

Chapter 5

Conclusions & Future Research

5.1 Conclusions

The two proposed simple activation function redundancy reduction approaches described in chapter 3 and demonstrated in chapter 4 show promising results. Both global and local methods work rather well, a substantial speedup can be achieved.

Redundancy reduction by the global approach is a bit slow. This makes the method only practical for small neural networks. The method does offer a behavior invariant transformation which can be controlled by user–specifications. A problem arises when the user–specification is wide. In case of a wide specification, the method tend to be too optimistic at the start of the reduction process. This undesirable optimism can result in a solution which is not necessarily cheaper to evaluate despite its enhanced redundancy.

The local approach does not have these disadvantages. It is fast, but does not guarantee a behavior invariant transformation, nor can this reduction process be easily controlled by user–specifications. The main reason is that the recorded local data depends on the behavior of other neurons connected in the previous layer. This means that controlling the redundancy reduction e.g. by back–propagating the specifications or automatic adaptation of the switching domains by user–specifications is not possible.

After the reduction process the output neurons usually turn up to be neurons with a linear activation function. This occurs, because to ensure a proper error convergence during training, the network output is restricted to be within 0.1 and 0.9. Linear output neurons may provide a better control of the reduction process by user–specification, more research is needed in that area.

The word size selection process is not very accurate. The integer part is never a problem. The determination if the required number of bits in the fractional part is not as easy. Apparently the information provided by the connection swing and Karnin pruning algorithm is not fine enough. Even the control of the connection pruning level can not be easily controlled by user–specifications. Similar problems arise with the local approach activation function selection process.

A connection with a large swing does not have to have a large Karnin sensitivity! So, both methods are somewhat incompatible. The Karnin algorithm is more accurate than the swing approach, because the total network's behavior in incorporated in the sensitivity data.

Finally we can conclude that a redundancy reduction process following the functional specifications in section 1.6 is not as easy as it seems. The preliminary work presented in this thesis look promising but is not sufficient for production purposes.

5.2 Future Research

At lot of work has to be done before the redundancy reduction process is ready for production purposes. We propose a combination of the two redundancy reduction methods, resulting in a relatively fast behavior invariant algorithm which can be controlled by specifications. More elaborate error models need to be constructed. The assumption that output neurons often have a linear activation function can severely reduce the model's complexity.

Another interesting area of research is the construction of a more advanced pruning algorithm. The idea is to make the pruning level depending on the user–specifications. We feel there is a relationship between the connection precision and the active area in the activation function.

Several other optimizations are possible. Units that mimic the output of other units can be removed. After bit–pruning, weights connected to a particular neuron having the same weight can be represented by a single connection. This weight sharing can further reduce evaluation cost.

Chapter 6

References

General:

[1] S. Haykin, *Neural Networks a Comprehensive Foundation*, USA: Macmillan College Publishing Company, 1994.

[2] J.M. Zurada, *Introduction to Atrificial Neural Systems,* St. Paul, MN, USA: West Publishing Company, 1992.

[3] J.A.G. Nijhuis, *An Engineering Approach to Neural Network Design,* Phd dissertation, Nijmegen University, 1992.

[4] V. Vysniauskas, F.C.A. Groen, and B.J.A. Kröse, "The optimal number of learning samples and hidden units in function approximation with a feedforward network", *Technical Report: CS−93−15, University of Amsterdam Faculty of Computer Science and Mathematics,* Amsterdam, The Netherlands, 1993.

[5] J. Sietsma, and R.J.F. Dow, "Creating Artificial Neural Networks That Generalize", *Neural Networks,* Vol. 4, pp. 67 – 79, 1991.

[6] S.C. Huang, and Y.F. Huang, "Bounds on the Number of Hidden Neurons in Multilayer Perceptrons", *IEEE Transactions on Neural Networks,* Vol. 2, No. 1, pp. 47 – 55, Januari 1991.

Fault Tolerance:

[7] M.D. Emmerson, and R.I. Damper, "Determining and Improving the Fault Tolerance of Multilayer Perceptrons in a Pattern−Recognition Application", *IEEE Transactions on Neural Networks,* Vol. 4, No. 5, pp. 788 – 793, September 1993.

[8] D.S. Phatak, and I. Koren, "Complete and Partial Fault Tolerance of Feedforward Neural Nets", *IEEE Transactions on Neural Networks,* Vol. 6, No. 2, pp. 446 – 456, March 1995.

[9] L.A. Belfore II, and B.W. Johnson, "The fault–tolerance of neural networks", *The International Journal of Neural Networks,* Vol. 1, No. 1, pp. 24 – 41, Januari 1991.

[10] L.A. Belfore II, B.W. Johnson, and J.H. Aylor, "Modeling of Fault–Tolerance in Neural Networks",(in: *Proceedings of IEEE International Joint Conference on Neural Networks*), Vol. 1, pp. 325 – 328, Januari 1990.

[11] G. Swaminathan, S. Srinivasan, and S. Mitra, "Fault Tolerance in Neural Networks", (in: *Proceedings of IEEE International Joint Conference on Neural Networks*), Vol. 2, pp. 699 – 702, Januari 1990.

[12] E. Pasero, "Fault Tolerance in Analog Neural Networks", (in: *Proceedings of the International Conference on Artificial Neural Networks*), Espoo, Finland, pp. 1557 – 1560, June 1991.

[13] U. Rückert, and H. Surmann, "Tolerance of a Binary Associative Memory Towards Stuck–At Faults", (in: *Proceedings of the International Conference on Artificial Neural Networks*), Espoo, Finland, pp. 1195 – 1198, June 1991.

[14] F. Vallet, and Ph. Kerlirzin, "Robustness in Multi–Layer Perceptrons", (in: *Proceedings of the International Conference on Artificial Neural Networks*), Espoo, Finland, pp. 641 – 646, June 1991.

[15] B.E. Segee, and M.J. Carter, "Comparative Fault Tolerance of Parallel Distributed Processing Networks (debunking the Myth of Inherent Fault Tolerance)", *Internal report: Robotics Laboratory, Intelligent Structures Group, University of New Hampshire,* New Hampshire, USA.

[16] A. Siggelkow, J. Nijhuis, S. Neußer, and L. Spaanenburg, "Influence of Hardware Characteristics on the Performance of a Neural Systems", (in: *Proceedings of the International Conference on Artificial Neural Networks*), Espoo, Finland, pp. 697 – 702, June 1991.

Pruning:

[17] R. Reed, "Pruning Algorithms – A Survey", *IEEE Transactions on Neural Networks,* Vol. 4, No. 5, pp. 740 – 747, September 1993.

[18] E.D. Karnin, "A Simple Procedure for Pruning Back–Propagation Trained Neural Networks", *IEEE Transactions on Neural Networks,* Vol. 1, No. 1, pp. 239 – 242, June 1990.

[19] S. Santini, "The Bearable Lightness of Being: Reducing the Number of Weights in Backpropagation Networks", *Artificial Neural Networks 2,* pp. 139 – 142, 1992.

[20] D. Tsaptsinos, A.R. Mirzai, and J.R. Leigh, "Matching the topology of a neural net to a particular problem: Preliminary results using correlation analysis as a pruning tool", *Artificial Neural Networks 2,* pp. 957 – 959, 1992.

Learning With Limited Precision:

[21] Y. Xie, "Training Algorithms for Limited Precision Feedforward Neural Networks", *SEDAL Technical Report No. 1991–8–3,* 1991.

[22] L.M. Reyneri, and E. Filippi, "An Analysis on the Performance of Solicon Implementations of Backpropagation Algorithms for Artificial Neural Networks", *IEEE Transactions on Computers,* Vol. 40, No. 12, pp. 1380 – 1389, December 1991.

[23] R.C.F. Frye, E.A. Rietman, and C.C. Wong, "Back–Propagation Learning and Nonidealities in Analog Network Hardware", *IEEE Transactions on Neural Networks,* Vol. 2, No. 1, pp. 110 – 117, Januari 1991.

[24] M. Hoehfeld, and S.E. Fahlman, "Learning with Limited Numerical Precision Using the Cascade–Correlation Algorithm", *Internal Report: CMU–CS–91–130, School of Computer Science, Carnegie Mellon University,* Pittsburg, PA, USA, 1991.

[25] S. Sakaue, T. Kohda, H. Yamamoto, S. Maruno, and Y. Shimeki, "Reduction of Required Precision Bits for Back–Propagation Applied to Pattern Recognition", *IEEE Transactions on Neural Networks,* Vol. 4, No. 2, pp. 270 – 275, March 1993.

Sensitivity Analysis:

[26] M. Stevenson, R. Winter, and B. Widrow, "Sensitivity of Feedforward Neural Networks to Weight Errors", *IEEE Transactions on Neural Networks,* Vol. 1, No. 1, pp. 71 – 80, March 1990.

[27] C. Alippi, "Sensitivity of Artificial Neurons to Weights and Inputs Quantizations: 1. The input neurons case", (in: *Proceedings of World Congress On Neural Networks*), Portland, Oregon, Vol. IV, pp. 317 – 322, July 1993.

[28] C. Alippi, "Sensitivity of Artificial Neurons to Weights and Inputs Quantizations: 1. The hidden neurons case", (in: *Proceedings of World Congress On Neural Networks*), Portland, Oregon, Vol. IV, pp. 312 – 316, July 1993.

[29] C. Alippi, and L. Briozzo, "The impact of finite precision in the VLSI implementation of Neural Architectures for image processing", (in: *Proceedings of World Congress On Neural Networks*), San Diago, California, Vol. II, pp. 537 – 542, 1994.

[30] J.Y. Choi, and C.H. Choi, "Sensitivity Analysis of Multilayer Perceptron with Differentiable Activation Functions", *IEEE Transactions on Neural Networks,* Vol. 3, No. 1, pp. 101 – 107, Januari 1992.

[31] Y. Xie, and M.A. Jabri, "Analysis of the Effects of Quantization in Multilayer Neural Networks Using a Statistical Model", *IEEE Transactions on Neural Networks,* Vol. 3, No. 2, pp. 334 – 338, March 1992.

[32] J.L. Holt, and J.N. Hwang, "Finite Precision Error Analysis of Neural Network Hardware Implementations",, *IEEE Transactions on Computers,* Vol. 42, No. 3, pp. 281 – 290, March 1993.

[33] S.W. Piché, "Robustness of Feedforward Neural Networks", (in: *Proceedings of IEEE International Joint Conference on Neural Networks*), Baltimore, Maryland, Vol. II, pp. 346 – 351, June 1992.

[34] S.W. Piché, "The Effects of Weight Errors in Neural Networks", (in: *Proceedings of World Congress On Neural Networks*), Portland, Oregon, Vol. IV, pp. 559 – 565, July 1993.

[35] H. Withagen, "Reducing the Effect of Quantization by Weight Scaling", (in: *Proceedings of IEEE International Joint Conference on Neural Networks*), Vol. IV, pp. 2128 – 2130, June 1994.

Neural Network Applications:

[36] J.H. Stevens, *"Determination of sand–grain size–distribution using neural networks"*, Master's Thesis, Groningen University, 1996.

[37] J.A.G. Nijhuis, M.H. ter Brugge, et al, "Car License Plate Recognition with Neural Networks and Fuzzy Logic", (in: *Proceedings of IEEE International Joint Conference on Neural Networks*), 1995.

Chapter 7

Appendix: Software

This chapter contains the source code of the implementation of the algorithms discussed in this thesis. The software is composed of two modules, **misc** and **lowcost**. Both modules are written in ANSI–C and interface with the **InterAct** neural network tooling environment. We will discuss in short the available routines in the modules. For details, the reader is referred to the complete documentation in the source code.

7.1 Module: LowCost

Module **LowCost** contains the implementation of the algorithms presented in this thesis. Routines are present for analyzing neural networks, determining redundancy, activation function simplification, connection pruning and word size selection.

To perform the variety of operations, **LowCost** maintains an internal data structure as shown in figure 7.1. Data structure is constructed and initialized by **InitNeuronStat**. Each hidden and output neuron has an entry in the **neuronStat_t** array pointed to by **neuronStat**. Connections are recorded in the **connectionStat_t** array accessible through the **connectionStat** pointer of the particular neuron. This data structure is later filled by the network analysis **GetNeuronStat** function. Based on this data, neuron activation functions are selected, irrelevant connections pruned and the number of bits for connections determined.

The following functions are accessible for **InterAct** application programs:

InitNeuronStat:
Routine builds the **neuronStat** and **connectionStat** administration shown in figure 7.1.

NeuronStatCleanup:
Routine frees space allocated by **InitNeuronStat**.

GetNeuronStat:
Routine analysis the neurons in the network and records things like minimum sums, maximum neuron output values, connection products etc.

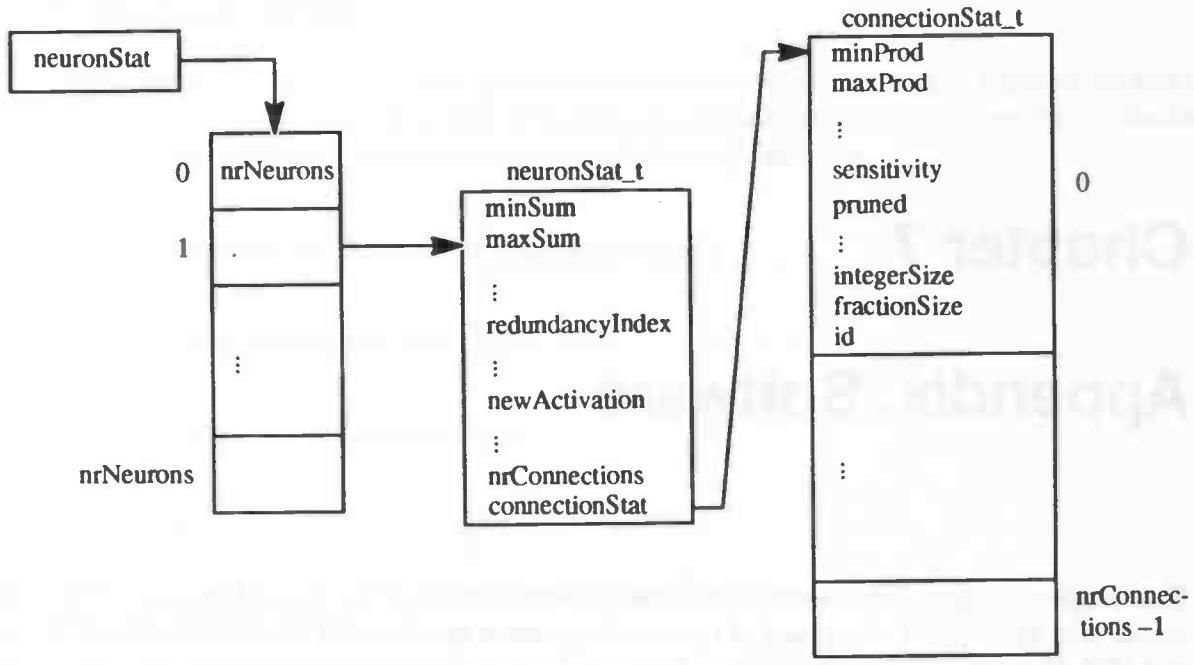


Figure 7.1: *Schematic view of LowCost's datastructure*

GetRedundancyIndex:
Routine calculates the redundancy index of a user designated neuron based on data recorded during the **GetNeuronStat** process.

WriteMatlabDatafile:
Dump the input output mapping of a neurons in a user specified data file. This file can later be processed by the **MatLab** package for further analysis or for producing neat graphs.

PrintNeuronStat:
This routine prints the internal data structure to a user specified file. The following example shows a part the output produced:

```
Activation function regions:
          oooo
        o
       o
      o
oooo
 0| 1| 2|3 |4
-------------------------
Neuron: 80
 Region Statistics:
  region: 0  frequency: 0
  region: 1  frequency: 1
  region: 2  frequency: 34
```

```
 region: 3  frequency: 15
 region: 4  frequency: 0
minSum: –1.530402  maxSum: 1.928731  inputSwing: 3.459133
minOut: 0.177935  maxOut: 0.873109  outputSwing: 0.695174
avgOut: 0.599438
oldBias: 1.115070  redundancy index: 19.633625
nrConnections: 6
LSQavgsum: 0.502135  LSQavgout: 0.599438
New Status: LINEAR with angle: 0.210366 and newBias: 3.462440
Connection Statistics:
 from: 2H  pruned: Y
      initialWeight: –2.144139  oldWeight: –2.200516  weight: –2.200516
      deltaWeight: 0.000518
      minProd: –0.314463  maxProd: –0.299097  avgProd: –0.306727
      swing: 0.015365  sensitivity: 0.575753
      total connection bit size: 0
 .....
 from: 7H  pruned: N
      initialWeight: –1.194525  oldWeight: –4.788313  weight: –4.788313
      deltaWeight: 0.002829
      minProd: –4.015392  maxProd: –0.002686  avgProd: –0.977154
      swing: 4.012705  sensitivity: 0.484040
      #bits integer part: 4  #bits fractional part: 4
      total connection word size: 8 bits
```

BuildConnectionSensitivity:
Prior to a BP learning cycle (epoch) this routine has be called. It gathers connection related data needed to perform a Karnin sensitivity analysis. After training, the accompanied function **AdjustConnectionSensitivity** is needed to adjust the recorded data.

AdjustConnectionSensitivity:
This routine adjusts the recorded data gathered by **BuildConnectionSensitivity.** This post learning process assures the Karnin sensitivity data is valid.

KarninConnectionPruning:
Remove a connection if the recorded sensitivity is less than **KARNIN_PRUNING_LEVEL**. The bias of the receiving neuron is adjusted accordingly. If a neuron looses all connections, it is removed from the network.

SwingConnectionPruning:
Remove a connection if the recorded swing is less than **SWING_PRUNING_LEVEL**. The bias of the receiving neuron is adjusted accordingly. If a neuron looses all connections, it is removed from the network.

SimplifyWeights:
Routine determines the required number of bits for the integer and fractional part of a connection.

SimplifyNeurons:
Routine determines a proper replacement for neuron activation function as discussed in section 3.5.2.

NetworkAnalysis:
This routine acts like a placeholder for the other simplification routines. It performs all necessary steps in the right order. Most **InterAct** applications only use this particular routine.

7.2 Module: Misc

Module **Misc** is a collection of some general purpose routines which are put together for convenience. The module has no internal data structures. It contains a collection of some handy macro's for determining **min, max** and **square,** and the following functions:

MyError:
General error printing routine, also terminates program.

MyMalloc:
Malloc, including a necessary size check.

MyCalloc:
Calloc, including a necessary size check.

EPrintf:
Printf to stderr.

ECPrintf:
Conditional printf to stderr.

7.3 Sources

See next page.

```
/**
 **      FILE            : lowcost.h
 **      VERSION         : 2.0
 **      DATE            : 11-02-95
 **      AUTHOR          : H.Keegstra
 **                        csg675@cs.rug.nl
 **
 **      DESCRIPTION     : InterAct application module, module contains routines
 **                        for analyzing neural networks, determining and exploiting
 **                        network redundancy in order to simplify activation functions
 **                        and determining proper connection word sizes.
 **/

#ifndef _LOW_COST_H_
#define _LOW_COST_H_

#include <stdio.h>
#include "interact.h"

/* Data structure holds data of a connection */
typedef struct connectionStat_t
{
  float minProd;        /* minimum connection product */
  float maxProd;        /* maximum connection product */
  float avgProd;        /* avgerage connection product */
  float initialWeight;  /* initial weight */
  float deltaWeight;    /* last BP weight update */
  float weight;         /* current weight */
  float oldWeight;      /* used to calculate deltaWeight, currently obsolete */
  float sensitivity;    /* Karnin sensitivity */
  bool pruned;          /* pruned (Y/N) */
  int integerSize;      /* weight integer part #bits */
  int fractionSize;     /* weight fractional part #bits */
  neuron_Sid_t id;      /* connected to */
} connectionStat_t;

/* Activation function replacement type */
typedef enum activationReplace_t
{
  sigmoid = 0,
  threshold,
  linear,
  hardlimiter,
  pruned
} activationReplace_t;


#define MAX_REGIONS 5

/* Data structure holds data of a neuron */
typedef struct neuronStat_t
{
  float minSum;                     /* minimum pre synaptic */
  float maxSum;                     /* maximum pre synaptic */
  float minOut;                     /* minimum post synaptic */
  float maxOut;                     /* maximum post synaptic */
  float LSQMinSum;                  /* minimum pre synaptic for LSQ slope det. */
  float LSQMaxSum;                  /* maximum pre synaptic for LSQ slope det. */
  float LSQMinOut;                  /* minimum post synaptic for LSQ slope det. */
  float LSQMaxOut;                  /* maximum post synaptic for LSQ slope det. */
  /* ---- */
  float avgSum;                     /* average pre synaptic */
  float avgOut;                     /* average post synaptic */
  float LSQAvgSum;                  /* average pre synaptic for LSQ slope det. */
  float LSQAvgOut;                  /* average post synaptic for LSQ slope det. */
  float LSQSumSum;                  /* intermediate for LSQ slope det. */
  float LSQSumOut;                  /* intermediate for LSQ slope det. */
  float LSQSumSquaredSum;           /* intermediate for LSQ slope det. */
  float LSQSumSumOut;               /* intermediate for LSQ slope det. */
```



```
  /* ---- */
  int nrLSQPatterns;                /* #patterns in LSQ slope */
  float redundancyIndex;            /* here it is, the redundancy index */
  float angle;                      /* the angle of the slope (=0, if not used) */
  float oldBias;                    /* old bias value */
  float bias;                       /* current bias value */
  activationReplace_t newActivation; /* new sctivation function type */
  long regionCount[MAX_REGIONS];    /* region statistic, used by PrintNeuronStat */
  long nrConnections;               /* nr. of inputs */
  connectionStat_t *connectionStat; /* the connections */
} neuronStat_t;


void GetRedundancyIndex(neuronStat_t *neuron);
/**
 ** Determine the redundancy index of the neuron designated by the neuron
 ** pointed to by neuron pointer.
 **/

neuronStat_t **InitNeuronStat(group_Sid_t neuronList);
/**
 ** Routine builds the neuronStat and connectionStat administration for neurons
 ** registered in neuronList.
 **/

void NeuronStatCleanUp(neuronStat_t ***neuronStat);
/**
 ** Routine frees space allocated by InitNeuronStat. POST: neuronStat = NULL.
 **/

#ifdef NO_PATTERN_LIST
/* Yes, some people don't like to use InterAct's pattern lists */
/* values taken from recog.c */
#define MAX_NOF_INPUTS 50
#define MAX_NOF_PATTERNS 5000
  int GetNeuronStat(neuronStat_t **neuronStat, group_Sid_t neuronList,
                    float patternList[MAX_NOF_PATTERNS][MAX_NOF_INPUTS],
                    long patternListSize, long nrPatternInput);
  /**
   ** Routine analysis the neurons registered in the neuronList by user supplied
   ** patterns. During the analysis phase several neuron and connection attributes
   ** are determined, which are stored in the neuronStat and connectionStat
   ** administration. The attributes are described in lowcost.h.
   **/

  void WriteMatlabDataFile(float patternList[MAX_NOF_PATTERNS][MAX_NOF_INPUTS],
                           long patternListSize, long nrPatternInput, char *preamble,
                           neuron_Sid_t id);
  /**
   ** Routine dumps the input output mapping of a neuron identified by id to a file.
   ** The data can be processed by the MatLab package to produce neat graphs.
   **/

  void NetworkAnalysis(float patternList[MAX_NOF_PATTERNS][MAX_NOF_INPUTS],
                       long patternListSize, long nrPatternInput, char *dFileName,
                       neuronStat_t **neuronStat);
  /**
   ** Routine acts as a placeholder for the optimization routines in this module.
   ** A application calls this routine when optimizations are required. If dFileName
   ** can not be opened an error is printed and the program terminated.
   **/
#else
  int GetNeuronStat(neuronStat_t **neuronStat, group_Sid_t neuronList,
                    pattern_Slist_id_t listId);
  /**
   ** Routine analysis the neurons registered in the neuronList by user supplied
   ** patterns. During the analysis phase several neuron and connection attributes
   ** are determined, which are stored in the neuronStat and connectionStat
   ** administration. The attributes are described in lowcost.h.
```

```
   **/

  void WriteMatlabDataFile(pattern_Slist_id_t listId, char *preamble, neuron_Sid_t id);
  /**
   ** Routine dumps the input output mapping of a neuron identified by id to a file.
   ** The data can be processed by the MatLab package to produce neat graphs.
   **/

  void NetworkAnalysis(pattern_Slist_id_t listId, char *dFileName,
                       neuronStat_t **neuronStat);
  /**
   ** Routine acts as a placeholder for the optimization routines in this module.
   ** A application calls this routine when optimizations are required.
   ** If dFileName can not be opened an error is printed and the program terminated.
   **/
#endif

int BuildConnectionSensitivity(neuronStat_t **neuronStat, group_Sid_t neuronList);
/**
 ** Routine performs a pre learning process used by the Karnin sensitivity analysis
 ** for the neurons registered in neuronList. Prior to a learning cycle (epoch) this
 ** routine must be called. After training phase, the accompanied routine
 ** AdjustConnectionSensitivity is needed.
 **/

int AdjustConnectionSensitivity(neuronStat_t **neuronStat, group_Sid_t neuronList,
                                float eta);
/**
 ** Routine performs a post learning process used by the Karnin sensitivity analysis for
 ** neurons registered in neuronList. The produce usefull results, prior to the learning
 ** phase, the BuildConnectionSensitivity routine should have been called.
 **/

int PrintNeuronStat(neuronStat_t **neuronStat, group_Sid_t neuronList, FILE *dataFile);
/**
 ** Routine prints the neuronStat and connectionStat contents of the neurons registered
 ** in the neuronList to the dataFile.
 **/

/* this is a preliminary value */
#define KARNIN_PRUNING_LEVEL 0.1f

int KarninConnectionPruning(neuronStat_t **neuronStat, group_Sid_t neuronList);
/**
 ** Routine performs the Karnin sensitivity pruning algorithm for neurons in neuronList.
 ** All connections having a sensitivity below KARNIN_PRUNING_LEVEL are removed, the
 ** bias of the receiving neurons are adjusted.
 ** PRE: BuildConnectionSensitivity & AdjustConnectionSensitivity are performed.
 **/

/* this is a preliminary value */
#define SWING_PRUNING_LEVEL 0.05f

int SwingConnectionPrunig(neuronStat_t **neuronStat, group_Sid_t neuronList);
/**
 ** Perform swing connection pruning algorithm on neurons registered in neuronList.
 ** All connections having a swing below SWING_PRUNING_LEVEL are removed, the bias
 ** of the reveiving neuron is adjusted.
 **/

int SimplifyWeights(neuronStat_t **neuronStat, group_Sid_t neuronList);
/**
 ** The bit-pruning algorithm. Determine number of required bits for the connections
 ** of the neurons registered in neuronList.
 **/

int SimplifyNeurons(neuronStat_t **neuronStat, group_Sid_t neuronList);
/**
 ** Routines simplifies the activation function of the neurons registered in neuronList.
```



```
 ** The selection process is based on a value called a redundancy index. After
 ** simplification several optimizations are performed concerning the angle and biases.
 **/

#endif
```

```
/**
 **      FILE            : lowcost.c
 **      VERSION         : 2.0
 **      DATE            : 11-02-95
 **      AUTHOR          : H.Keegstra
 **                        csg675@cs.rug.nl
 **
 **      DESCRIPTION     : InterAct application module, module contains routines
 **                        for analyzing neural networks, determining and exploiting
 **                        network redundancy in order to simplify activation functions
 **                        and determining proper connection word sizes.
 **/

#include <math.h>
#include <stdio.h>
#include <ctype.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <time.h>
#include <limits.h>
#include <float.h>
#include "interact.h"
#include "misc.h"
#include "new.h"
#include "interactmacros.h"

/* MACRO checks parameters */
#define CHECK_PARA ( \
                    if(neuronStat == NULL) \
                      MyError("neuronStat has not been initialized\n"); \
                    if(neuronList == input_Slist) \
                      return -1; \
                  )

/* MACRO checks neuronStat administration */
#define CHECK_ID(id) ( \
                      get_Sneuron_kind(id, &neuronKind, &status); \
                      switch(neuronKind) \
                      ( \
                        case kind_Shidden_neuron: \
                        case kind_Soutput_neuron: \
                          break; \
                        default: \
                          get_Sid_neuron(get_Soption_larger, id, neuronList, "", &id,\
                                         &neuronListStatus); \
                          continue; \
                      ) \
                      if(id == 0) \
                        MyError("Oops! Neuron 0 is not an input neuron, now I have a pr
oblem!\n"); \
                      if(neuronStat[id] == NULL) \
                        MyError("Interact administration error, unknown neuronId: %ld\n
", id); \
                    )

/*
      if(neuronStat[id]->newActivation == pruned) \
                      MyError("Interact administration error, pruned neuron in neuron
List: %ld\n",id); \
*/

/* MACRO calculate standard logistic activation function */
#define F(sum) (1.0f / (1.0f + exp(-(float)(sum))))

/* MACRO calculate inverse of standard logistic activation function */
#define FINV(out) (-log(1.0f - (float)(out)) / (float)(out))
```



```
/* MACRO calculate derivative of standard logistic activation function */
#define DF(sum) (1.0f / (exp((float)(sum)) * (1 + exp(-(float)(sum))) * (1 + exp(-(float
)(sum)))))

/* preliminary values, determine post synaptic range for LSQ slope det. */
#define LSQ_MIN 0.05f
#define LSQ_MAX 0.95f

void GetRedundancyIndex(neuronStat_t *neuron)
/**
 ** IN/OUT: neuron = pointer to neuronStat neuron administration
 **
 ** DESCR:
 ** Determine the redundancy index of the neuron designated by the neuron
 ** pointed to by neuron pointer.
 **/
{
  float error;
  float sumSwing = fabs(neuron->maxSum - neuron->minSum);

  /* prevent log(0) runtime errors */
  if(neuron->minOut / neuron->maxOut == 0.0f)
  {
    neuron->redundancyIndex = 0.0f;
    return;
  }

  error = (neuron->avgOut - 1.0f) * (neuron->avgOut - 1.0f) * sumSwing +
          (1.0f - 2.0f * neuron->avgOut) * log(neuron->minOut / neuron->maxOut) +
          neuron->minOut - neuron->maxOut;

  neuron->redundancyIndex =  sumSwing / error;
}


static char GetNeuronKindChar(neuron_Sid_t id)
/**
 ** IN: id = id of neuron to be processed
 **
 ** RET: char indentifying neuron type:
 **        'I' = input neuron
 **        'H' = hidden neuron
 **        'O' = output neuron
 **
 ** DESCR:
 ** Routine determines the type of neuron id, this type (char) is used by
 ** output generating routines, PrintNeuronStat etc.
 **/
{
  status_St      status;
  neuron_Skind_t neuronKind;

  get_Sneuron_kind(id, &neuronKind, &status);
  switch(neuronKind)
  {
    case kind_Sinput_neuron:
      return 'I';
    case kind_Shidden_neuron:
      return 'H';
    case kind_Soutput_neuron:
      return 'O';
    default:
      return '?';
  }
}


#ifdef NO_PATTERN_LIST
  void WriteMatlabDataFile(float patternList[MAX_NOF_PATTERNS][MAX_NOF_INPUTS],
```

```
                        long patternListSize, long nrPatternInput, char *preamble,
                        neuron_Sid_t id)
/**
 ** IN: patternList = patternList used by OCR application
 **     patternListSize = nr. of patterns in patternList
 **     nrPatternInput = nr. of inputs to the network, needed for copy process
 **/
#else
  void WriteMatlabDataFile(pattern_Slist_id_t listId, char *preamble, neuron_Sid_t id)
/**
 ** IN: listId = pattern list designator
 **
 **/
#endif
/**
 ** IN: preamble = string designating the output file name and or its path
 **     id = id of the neuron the processed
 **
 ** DESCR:
 ** Routine dumps the input output mapping of a neuron identified by id to a file.
 ** The data can be processed by the MatLab package to produce neat graphs.
 **/
{
  status_St             status;
  pattern_Slist_info_t  info;
  FILE                  *matlabFile;
  char                  matlabFileName[255];
  long                  nrInputs, nrTargets, nrPatterns, patternId;
  out_St                output;
  get_Sneuron_status_t  neuronStatus;
  neuron_Skind_t        neuronKind;
  out_St                inputs[INTERACT_MAX_PATTERN_INPUTS];
  out_St                targets[INTERACT_MAX_PATTERN_INPUTS];

  get_Sneuron_kind(id, &neuronKind, &status);
  if(neuronKind == kind_Sinput_neuron)
    return;

  if(strlen(preamble) > 200)
    MyError("WriteMatlabDataFile: parameter preamble is too long,\n it should not exceed
 200 chars.\n");
  strcpy(matlabFileName, preamble);
  sprintf(matlabFileName + strlen(preamble), "%c%ld.dat", GetNeuronKindChar(neuronKind),
 id);
  if((matlabFile = fopen(matlabFileName, "w")) == NULL)
    MyError("%s could not be opened for write\n", matlabFileName);

  #ifdef NO_PATTERN_LIST
    nrPatterns = patternListSize;
  #else
    pattern_Slist_get_info(listId, &info, &status);
    nrPatterns = info.nr_patterns;
  #endif
  for(patternId = 1; patternId <= nrPatterns; patternId++)
  {
    #ifdef NO_PATTERN_LIST
    {
      out_St inX[MAX_NOF_INPUTS];
      long i;

      for(i = 0; i < nrPatternInput; i++) /* this will slow things down! */
        inX[i] = (out_St)patternList[patternId][i];
      slff_evalu_pattern(inX, nrPatternInput);
    }
    #else
      pattern_Slist_get_pattern(listId, patternId, inputs, &nrInputs, targets,
                                &nrTargets, &status);
      set_Sinput(inputs, nrInputs, &status);
      calc_Sstart_evalu(0L, hidden_Slist, &status);
```



```
      calc_Sstart_evalu(0L, output_Slist, &status);
    #endif
    get_Sstatus_neuron(id, &neuronStatus, &status);
    get_Sneuron_output(id, &output, &status);
    fprintf(matlabFile, "%f %f\n", neuronStatus.sum, output);
  }
  fclose(matlabFile);
}


neuronStat_t **InitNeuronStat(group_Sid_t neuronList)
/**
 ** IN: neuronList = InterAct list containing neurons, e.g. hidden_Slist
 **
 ** RET: pointer to neuronStat administration
 **
 ** DESCR:
 ** Routine builds the neuronStat and connectionStat administration for neurons
 ** registered in neuronList.
 **/
{
  status_St          status, neuronListStatus;
  neuron_Sid_t       id;
  neuron_Skind_t     neuronKind;
  long               nrNeurons;
  neuronStat_t       **neuronStat;
  get_Sinput_info_t  connections[INTERACT_MAX_INPUTS];
  long               nrConnections;
  long               i;

  if(neuronList != neuron_Slist)
    MyError("InitNeuronStat needs neuron_Slist\n");

  get_Snumber_of_neurons(neuronList, &nrNeurons, &status);
  neuronStat = (neuronStat_t**)MyCalloc((nrNeurons + 1)* sizeof(neuronStat_t*));
  /* neuron 0 is always an input neuron and is subsequently not needed */
  /* in the neuronStat administration, so we can use it as a sentinel */
  neuronStat[0] = (neuronStat_t*)nrNeurons;

  get_Sid_neuron(get_Soption_first, 0L, neuronList, "", &id, &neuronListStatus);
  while(neuronListStatus.all == status_Sok)
  {
    get_Sneuron_kind(id, &neuronKind, &status);
    switch(neuronKind)
    {
      case kind_Shidden_neuron:
      case kind_Soutput_neuron:
        break;
      default:
        get_Sid_neuron(get_Soption_larger, id, neuronList, "", &id, &neuronListStatus);
        continue;
    }
    if(id == 0)
      MyError("Oops! Neuron 0 is not an input neuron, now I have a problem!\n");
    if(neuronStat[id] != NULL)
      MyError("Interact administration error, unknown neuronId: %ld\n", id);

    neuronStat[id] = (neuronStat_t*)MyCalloc(sizeof(neuronStat_t));
    neuronStat[id]->minSum = FLT_MAX;
    neuronStat[id]->maxSum = -FLT_MAX;
    neuronStat[id]->minOut = FLT_MAX;
    neuronStat[id]->maxOut = -FLT_MAX;

    get_Sneuron_inputs(id, connections, &nrConnections, &status);
    neuronStat[id]->nrConnections = nrConnections;

    neuronStat[id]->connectionStat = (connectionStat_t*)MyCalloc(nrConnections * sizeof(
connectionStat_t));
```

```
    for(i = 0; i < nrConnections; i++)
    {
      /* help compiler to perform CSE */
      connectionStat_t *c = &neuronStat[id]->connectionStat[i];

      c->id = connections[i].from_id;
      c->initialWeight = c->oldWeight = c->weight = connections[i].weight;
      c->minProd = FLT_MAX;
      c->maxProd = -FLT_MAX;
    }
    get_Sid_neuron(get_Soption_larger, id, neuronList, "", &id, &neuronListStatus);
  }
  return neuronStat;
}


void NeuronStatCleanUp(neuronStat_t ***neuronStat)
/**
 ** IN/OUT: neuronStat = pointer to pointer to neuronStat administration to be removed
 **
 ** DESCR:
 ** Routine frees space allocated by InitNeuronStat, POST: neuronStat = NULL.
 **/
{
  int i;
  int size = (int)(*neuronStat); /* we use location 0 as a sentinel */

  if(*neuronStat == NULL)
    return;

  for(i = 1; i < size; i++)
  {
    if((*neuronStat)[i] != NULL)
    {
      free((*neuronStat)[i]->connectionStat);
      free((*neuronStat)[i]);
    }
  }
  free(*neuronStat);
  neuronStat = NULL;
}


/* in case of logistic activation function (a = 1) */
#define REGION_0_LEVEL -2.997
#define REGION_1_LEVEL -1.5
#define REGION_2_LEVEL -REGION_1_LEVEL
#define REGION_3_LEVEL -REGION_0_LEVEL

#ifdef NO_PATTERN_LIST
  int GetNeuronStat(neuronStat_t **neuronStat, group_Sid_t neuronList,
                    float patternList[MAX_NOF_PATTERNS][MAX_NOF_INPUTS],
                    long patternListSize, long nrPatternInput)
/**
 ** IN: patternList = patternList used by OCR application
 **     patternListSize = nr. of patterns in patternList
 **     nrPatternInput = nr. of inputs to the network, needed for copy process
 **/
#else
  int GetNeuronStat(neuronStat_t **neuronStat, group_Sid_t neuronList,
                    pattern_Slist_id_t listId)
/**
 ** IN: listId = pattern list designator
 **/
#endif
/**
 ** IN: neuronList = InterAct neuron list, e.g. hidden_Slist
 **
 ** IN/OUT: neuronStat = pointer to neuronStat administration
```



```
 **
 ** RET: -1 if error, 0 otherwise
 **
 ** DESCR:
 ** Routine analysis the neurons registered in the neuronList by user supplied
 ** patterns. During the analysis phase several neuron and connection attributes
 ** are determined, which are stored in the neuronStat and connectionStat
 ** administration. The attributes are described in lowcost.h.
 **/
{
  status_St             status, neuronListStatus;
  long                  nrPatterns, patternId;
  pattern_Slist_info_t  info;
  neuron_Sid_t          id;
  neuron_Skind_t        neuronKind;
  out_St                output;
  get_Sneuron_status_t  neuronStatus;
  long                  i;
  out_St                inputs[INTERACT_MAX_PATTERN_INPUTS];
  out_St                targets[INTERACT_MAX_PATTERN_INPUTS];
  long                  nrInputs, nrTargets;


  CHECK_PARA;

  /* clear areas, just te be sure */
  get_Sid_neuron(get_Soption_first, 0L, neuronList, "", &id, &neuronListStatus);
  while(neuronListStatus.all == status_Sok)
  {
    CHECK_ID(id);

    neuronStat[id]->LSQMinSum = neuronStat[id]->minSum = FLT_MAX;
    neuronStat[id]->LSQMaxSum = neuronStat[id]->maxSum = -FLT_MAX;
    neuronStat[id]->LSQMinOut = neuronStat[id]->minOut = FLT_MAX;
    neuronStat[id]->LSQMaxOut = neuronStat[id]->maxOut = -FLT_MAX;
    neuronStat[id]->nrLSQPatterns = 0;
    /* clear all summing fields */
    /* please watch field order in neuronStat_t structure! */
    memset(neuronStat[id] + 8 * sizeof(float), 0, 8 * sizeof(float));

    get_Sstatus_neuron(id, &neuronStatus, &status);
    neuronStat[id]->oldBias = neuronStatus.bias;

    memset(neuronStat[id]->regionCount, 0, MAX_REGIONS * sizeof(long));

    for(i = 0; i < neuronStat[id]->nrConnections; i++)
    {
      neuronStat[id]->connectionStat[i].minProd = FLT_MAX;
      neuronStat[id]->connectionStat[i].maxProd = -FLT_MAX;
      neuronStat[id]->connectionStat[i].avgProd = 0.0f;
    }
    get_Sid_neuron(get_Soption_larger, id, neuronList, "", &id, &neuronListStatus);
  }

#ifdef NO_PATTERN_LIST
  nrPatterns = patternListSize;
#else
  pattern_Slist_get_info(listId, &info, &status);
  nrPatterns = info.nr_patterns;
#endif

for(patternId = 1; patternId <= nrPatterns; patternId++)
{
  #ifdef NO_PATTERN_LIST
  {
    out_St inX[MAX_NOF_INPUTS];

    for(i = 0; i < nrPatternInput; i++)
      inX[i] = (out_St)patternList[patternId][i];
```

```
      slff_evalu_pattern(inX, nrPatternInput);
    )
    #else
      pattern_Slist_get_pattern(listId, patternId, inputs, &nrInputs, targets,
                                &nrTargets, &status);
      set_Sinput(inputs, nrInputs, &status);
      calc_Sstart_evalu(0L, hidden_Slist, &status);
      calc_Sstart_evalu(0L, output_Slist, &status);
    #endif

    get_Sid_neuron(get_Soption_first, 0L, neuronList, "", &id, &neuronListStatus);
    while(neuronListStatus.all == status_Sok)
    {
      CHECK_ID(id);

      get_Sstatus_neuron(id, &neuronStatus, &status);
      get_Sneuron_output(id, &output, &status);
      neuronStat[id]->minSum = min(neuronStat[id]->minSum, neuronStatus.sum);
      neuronStat[id]->maxSum = max(neuronStat[id]->maxSum, neuronStatus.sum);
      neuronStat[id]->minOut = min(neuronStat[id]->minOut, output);
      neuronStat[id]->maxOut = max(neuronStat[id]->maxOut, output);
      neuronStat[id]->avgSum += neuronStatus.sum;
      neuronStat[id]->avgOut += output;
      if(output > LSQ_MIN && output < LSQ_MAX)
      {
        neuronStat[id]->nrLSQPatterns++;
        neuronStat[id]->LSQSumSum += neuronStatus.sum;
        neuronStat[id]->LSQSumSquaredSum += neuronStatus.sum * neuronStatus.sum;
        neuronStat[id]->LSQSumOut += output;
        neuronStat[id]->LSQSumSumOut += neuronStatus.sum * output;
        neuronStat[id]->LSQMinSum = min(neuronStat[id]->LSQMinSum, neuronStatus.sum);
        neuronStat[id]->LSQMaxSum = max(neuronStat[id]->LSQMaxSum, neuronStatus.sum);
        neuronStat[id]->LSQMinOut = min(neuronStat[id]->LSQMinOut, output);
        neuronStat[id]->LSQMaxOut = max(neuronStat[id]->LSQMaxOut, output);
      }
      /* build activation function region statistics */
      if(neuronStatus.sum < REGION_0_LEVEL)
        neuronStat[id]->regionCount[0]++;
      else
        if(neuronStatus.sum < REGION_1_LEVEL)
          neuronStat[id]->regionCount[1]++;
        else
          if(neuronStatus.sum < REGION_2_LEVEL)
            neuronStat[id]->regionCount[2]++;
          else
            if(neuronStatus.sum < REGION_3_LEVEL)
              neuronStat[id]->regionCount[3]++;
            else
              neuronStat[id]->regionCount[4]++;

      for(i = 0; i < neuronStat[id]->nrConnections; i++)
      {
        float output, prod;
        /* help compiler to perform CSE */
        connectionStat_t *c = &neuronStat[id]->connectionStat[i];

        if(c->pruned)
          continue;

        get_Sneuron_output(c->id, &output, &status);

        prod = output * c->weight;
        c->minProd = min(c->minProd, prod);
        c->maxProd = max(c->maxProd, prod);
        c->avgProd += prod / (float)nrPatterns;
      }
      get_Sid_neuron(get_Soption_larger, id, neuronList, "", &id, &neuronListStatus);
    }
  }
```



```
  get_Sid_neuron(get_Soption_first, 0L, neuronList, "", &id, &neuronListStatus);
  while(neuronListStatus.all == status_Sok)
  {
    CHECK_ID(id);

    neuronStat[id]->avgSum /= (float)nrPatterns;
    neuronStat[id]->avgOut /= (float)nrPatterns;
    neuronStat[id]->LSQAvgSum = neuronStat[id]->LSQSumSum / (float)neuronStat[id]->nrLSQ
Patterns;
    neuronStat[id]->LSQAvgOut = neuronStat[id]->LSQSumOut / (float)neuronStat[id]->nrLSQ
Patterns;
    GetRedundancyIndex(neuronStat[id]);
    get_Sid_neuron(get_Soption_larger, id, neuronList, "", &id, &neuronListStatus);
  }
  return 0;
}


int PrintNeuronStat(neuronStat_t **neuronStat, group_Sid_t neuronList, FILE *dataFile)
/**
 ** IN: neuronStat = pointer to neuronStat neuron administration
 **     neuronList = InterAct neuron list, e.g. hidden_Slist
 **
 ** OUT: dataFile = file receiving the output of this routine
 **
 ** RET: -1 if error, 0 otherwise
 **
 ** DESCR:
 ** Routine prints the neuronStat and connectionStat contents of the neurons registered
 ** in the neuronList to the dataFile.
 **/
{
  status_St       status, neuronListStatus;
  neuron_Sid_t    id;
  long            i;
  neuron_Skind_t neuronKind;
  int             nrPruned = 0;

  CHECK_PARA;

  fprintf(dataFile,"Activation function regions:\n");
  fprintf(dataFile,"          oooo\n");
  fprintf(dataFile,"        o     \n");
  fprintf(dataFile,"       o      \n");
  fprintf(dataFile,"      o       \n");
  fprintf(dataFile,"oooo          \n");
  fprintf(dataFile," 0 |1|2|3| 4 \n\n");

  fprintf(dataFile, "----------\n");
  fprintf(dataFile, "Neuron Statistics:\n\n");
  get_Sid_neuron(get_Soption_first, 0L, neuronList, "", &id, &neuronListStatus);
  while(neuronListStatus.all == status_Sok)
  {
    CHECK_ID(id);

    fprintf(dataFile, "Neuron: %ld%c\n", id, GetNeuronKindChar(id));
    fprintf(dataFile, "  Region Statistics:\n");
    for(i = 0; i < MAX_REGIONS ; i++)
      fprintf(dataFile, "    region: %ld  frequency: %ld\n", i,
              neuronStat[id]->regionCount[i]);
    fprintf(dataFile, "  minSum: %f  maxSum: %f  inputSwing: %f\n",
            neuronStat[id]->minSum, neuronStat[id]->maxSum,
            fabs(neuronStat[id]->maxSum - neuronStat[id]->minSum));
    fprintf(dataFile, "  minOut: %f  maxOut: %f  outputSwing: %f\n",
            neuronStat[id]->minOut, neuronStat[id]->maxOut,
            fabs(neuronStat[id]->maxOut - neuronStat[id]->minOut));
    fprintf(dataFile, "  avgOut: %f\n", neuronStat[id]->avgOut);
    fprintf(dataFile, "  oldBias: %f  redundancy index: %f\n", neuronStat[id]->oldBias,
            neuronStat[id]->redundancyIndex);
```

```
    fprintf(dataFile, "  nrConnections: %ld\n", neuronStat[id]->nrConnections);
    fprintf(dataFile, "  LSQavgsum: %f  LSQavgout: %f\n", neuronStat[id]->LSQAvgSum,
            neuronStat[id]->LSQAvgOut);

    fprintf(dataFile, "  New Status: ");
    switch(neuronStat[id]->newActivation)
    {
      case sigmoid:
        fprintf(dataFile, "SIGMOID (unchanged)\n");
        break;
      case threshold:
        fprintf(dataFile, "THRESHOLD with angle: %f and newBias: %f\n",
                neuronStat[id]->angle, neuronStat[id]->bias);
        break;
      case linear:
        fprintf(dataFile, "LINEAR with angle: %f and newBias: %f\n",
                neuronStat[id]->angle, neuronStat[id]->bias);
        break;
      case hardlimiter:
        fprintf(dataFile, "HARDLIMITER with newBias: %f\n", neuronStat[id]->bias);
        break;
      case pruned: /* this will never be executed */
        fprintf(dataFile, "PRUNED\n");
        break;
      default:
        MyError("Internal neuronStat administration error, unkown newActivation: %d\n",
                neuronStat[id]->newActivation);
    }

    fprintf(dataFile, "  Connection Statistics:\n");
    for(i = 0; i < neuronStat[id]->nrConnections; i++)
    {
      /* help compiler to perform CSE */
      connectionStat_t *c = &neuronStat[id]->connectionStat[i];

      if(c->pruned)
        nrPruned++;
      fprintf(dataFile, "    from: %ld%c  pruned: %c\n", c->id, GetNeuronKindChar(c->id)
,
              c->pruned ? 'Y' : 'N');
      fprintf(dataFile, "                initialWeight: %f  oldWeight: %f  weight: %f\n",
              c->initialWeight, c->oldWeight, c->weight);
      fprintf(dataFile, "                deltaWeight: %f\n", c->deltaWeight);
      fprintf(dataFile, "                minProd: %f  maxProd: %f  avgProd: %f\n",
              c->minProd, c->maxProd, c->avgProd);
      fprintf(dataFile, "                swing: %f  sensitivity: %f\n",
              fabs(c->maxProd - c->minProd), c->sensitivity);
      if(c->pruned)
        fprintf(dataFile, "                total connection bit size: 0\n");
      else
      {
        fprintf(dataFile, "                #bits integer part: %d  #bits fractional part: %d
\n", c->integerSize, c->fractionSize);
        fprintf(dataFile, "                total connection word size: %d bits\n",
                c->integerSize + c->fractionSize);
      }
    }
    get_Sid_neuron(get_Soption_larger, id, neuronList, "", &id, &neuronListStatus);
    fprintf(dataFile, "----------\n");
  }
  fprintf(dataFile, "\n  #Connections Pruned: %d\n", nrPruned);
  return 0;
}


int BuildConnectionSensitivity(neuronStat_t **neuronStat, group_Sid_t neuronList)
/**
 ** IN: neuronList = InterAct neuron list e.g. hidden_Slist
 **
```



```
 ** IN/OUT: neuronStat = pointer to neuronStat neuron administration
 **
 ** RET: -1 if error, 0 otherwise
 **
 ** DESCR:
 ** Routine performs a pre learning process used by the Karnin sensitivity analysis
 ** for the neurons registered in neuronList. After training phase, the accompanied
 ** routine: AdjustConnectionSensitivity is needed.
 **/
{
  status_St               status, neuronListStatus;
  neuron_Sid_t            id;
  long                    i;
  get_Sconnection_info_t connection;
  neuron_Skind_t          neuronKind;

  CHECK_PARA;

  get_Sid_neuron(get_Soption_first, 0L, neuronList, "", &id, &neuronListStatus);
  while(neuronListStatus.all == status_Sok)
  {
    CHECK_ID(id);

    connection.to_site = site_Sdefault_site;
    connection.to_id = id;
    for(i = 0; i < neuronStat[id]->nrConnections; i++)
    {
      /* help compiler to perform CSE */
      connectionStat_t *c = &neuronStat[id]->connectionStat[i];

      if(c->pruned)
        continue;
      connection.from_id = c->id;
      get_Sinfo_connection(&connection, &status);
      c->deltaWeight = fabs(c->weight - connection.weight);
      c->sensitivity += c->deltaWeight * c->deltaWeight;
      c->oldWeight = c->weight = connection.weight;
    }
    get_Sid_neuron(get_Soption_larger, id, neuronList, "", &id, &neuronListStatus);
  }
  return 0;
}


int AdjustConnectionSensitivity(neuronStat_t **neuronStat, group_Sid_t neuronList,
                                float eta)
/**
 ** IN: neuronList = InterAct neuron list, e.g. hidden_Slist
 **     eta = learning rate
 **
 ** IN/OUT: neuronStat = pointer to neuronStat neuron administration
 **
 ** RET: -1 if error, 0 otherwise
 **
 ** DESCR:
 ** Routine performs a post learning process used by the Karnin sensitivity analysis for
 ** neurons registered in neuronList. The produce usefull results, prior to the learning
 ** phase, the BuildConnectionSensitivity routine should have been called.
 **/
{
  status_St       status, neuronListStatus;
  neuron_Sid_t    id;
  long            i;
  neuron_Skind_t neuronKind;

  CHECK_PARA;

  get_Sid_neuron(get_Soption_first, 0L, neuronList, "", &id, &neuronListStatus);
  while(neuronListStatus.all == status_Sok)
```

```
  {
    CHECK_ID(id);

    for(i = 0; i < neuronStat[id]->nrConnections; i++)
    {
      /* help compiler to perform CSE */
      connectionStat_t *c = &neuronStat[id]->connectionStat[i];

      if(c->pruned)
        continue;
      c->sensitivity *= c->weight / (eta * (c->weight - c->initialWeight));
      c->sensitivity = fabs(c->sensitivity);
    }
    get_Sid_neuron(get_Soption_larger, id, neuronList, "", &id, &neuronListStatus);
  }
  return 0;
}


static int AdjustBias(neuronStat_t **neuronStat, group_Sid_t neuronList,
                      neuron_Sid_t curId, neuron_Sid_t fromId)
/**
 ** IN: neuronList = InterAct neuron list, e.g. hidden_Slist
 **     curId = current neuron id
 **     fromId = id of neuron from which a connection is coming
 **
 ** RET: -1 if error, 0 otherwise
 **
 ** IN/OUT: neuronStat = pointer to neuronStat neuron administration
 **
 ** DESCR:
 ** Routine adjusts the bias of curId neuron by the avgProd of connection
 ** fromId to curId.
 **/
{
  status_St status;
  float     bias;

  CHECK_PARA;

  if(neuronStat[curId]->newActivation == pruned)
    return -1;

  neuronStat[curId]->bias += neuronStat[curId]->connectionStat[fromId].avgProd *
                             neuronStat[curId]->connectionStat[fromId].weight;
  set_Sbias_random(curId, neuronList, -FLT_MAX, FLT_MAX, bias, 1.0, rand_Sdummy,
                   &status);
  return 0;
}


static int RemoveConnection(neuronStat_t **neuronStat, group_Sid_t neuronList,
                            neuron_Sid_t curId, neuron_Sid_t fromId)
/**
 ** IN: neuronList = InterAct neuron list, e.g. hidden_Slist
 **     curId = current neuron id
 **     fromId = id of neuron from which a connection is coming
 **
 ** IN/OUT: neuronStat = pointer to neuronStat neuron administration
 **
 ** RET: -1 if error, 0 otherwise
 **
 ** DESCR:
 ** Routine removes connection from fromId to curId. If the number of input
 ** connections is zero,  the Routine removed the neuron. NOTE: this only works
 ** for 3-layer networks.
 **/
{
  status_St         status;
```



```
  get_Sinput_info_t inputConnections[INTERACT_MAX_INPUTS];
  neuron_Sid_t      fromIdOutputConnections[INTERACT_MAX_OUTPUTS];
  neuron_Skind_t    neuronKind;
  long              nrInputs;
  long              nrFromIdOutputs;

  CHECK_PARA;

  get_Sneuron_kind(curId, &neuronKind, &status);
  if(neuronKind == kind_Sinput_neuron)
    return -1;
  if(neuronStat[curId]->connectionStat[fromId].pruned)
    return -1;

  AdjustBias(neuronStat, neuronList, curId, fromId);
  connection_Sdelete(fromId, curId, neuronList, neuronList, site_Sdefault_site,
                     connection_Ssingle, &status);
  neuronStat[curId]->connectionStat[fromId].pruned = TRUE;

  /* remove neuron if #connections == 0 */
  if(neuronKind == kind_Shidden_neuron)
  {
    get_Sneuron_inputs(curId, inputConnections, &nrInputs, &status);
    if(nrInputs == 0 && neuronKind != kind_Soutput_neuron)
    {
      /* also update bias of connected neurons */
      get_Sneuron_outputs(curId, fromIdOutputConnections, &nrFromIdOutputs, &status);
      while(--nrFromIdOutputs >= 0)
        AdjustBias(neuronStat, neuronList, fromIdOutputConnections[nrFromIdOutputs],
                   curId);
      neuronStat[curId]->newActivation = pruned;
      neuron_Sdelete(curId, neuronList, &status);
    }
  }
  else /* curId == kind_Soutput_neuron */
  {
    get_Sneuron_outputs(fromId, fromIdOutputConnections, &nrFromIdOutputs, &status);
    if(nrFromIdOutputs == 0)
    {
      neuron_Sdelete(fromId, neuronList, &status);
      neuronStat[curId]->newActivation = pruned;
    }
  }
  return 0;
}


int KarninConnectionPruning(neuronStat_t **neuronStat, group_Sid_t neuronList)
/**
 ** IN: neuronList = InterAct neuron list, e.g. hidden_Slist
 **
 ** IN/OUT: neuronStat = pointer to neuronStat neuron administration
 **
 ** RET: number of connections pruned, -1 if error
 **
 ** DESCR:
 ** Routine performs the Karnin sensitivity pruning algorithm for neurons in neuronList.
 ** All connections having a sensitivity below KARNIN_PRUNING_LEVEL are removed,
 ** the bias of the receiving neurons are adjusted.
 ** PRE: BuildConnectionSensitivity & AdjustConnectionSensitivity are performed.
 **/
{
  status_St      status, neuronListStatus;
  neuron_Sid_t   id;
  long           i;
  neuron_Skind_t neuronKind;
  int            nrPruned = 0;

  CHECK_PARA;
```

```
  get_Sid_neuron(get_Soption_first, 0L, neuronList, "", &id, &neuronListStatus);
  while(neuronListStatus.all == status_Sok)
  {
    CHECK_ID(id);

    for(i = 0; i < neuronStat[id]->nrConnections; i++)
    {
      /* help compiler to perform CSE */
      connectionStat_t *c = &neuronStat[id]->connectionStat[i];

      if(c->pruned)
        continue;
      if(c->sensitivity <= KARNIN_PRUNING_LEVEL)
      {
        RemoveConnection(neuronStat, neuronList, id, c->id);
        nrPruned++;
      }
    }
    get_Sid_neuron(get_Soption_larger, id, neuronList, "", &id, &neuronListStatus);
  }
  return nrPruned;
}


int SwingConnectionPrunig(neuronStat_t **neuronStat, group_Sid_t neuronList)
/**
 ** IN: neuronList = InterAct neuron list, e.g. hidden_Slist
 **
 ** IN/OUT: neuronStat = pointer to neuronStat neuron administration
 **
 ** RET: number of connections pruned, -1 if error
 **
 ** DESCR:
 ** Perform swing connection pruning algorithm on neurons registered in neuronList.
 ** All connections having a swing below SWING_PRUNING_LEVEL are removed, the bias
 ** of the reveiving neuron is adjusted.
 **/
{
  status_St      status, neuronListStatus;
  neuron_Sid_t   id;
  long           i;
  neuron_Skind_t neuronKind;
  int            nrPruned = 0;
  float          pruneSwing;

  CHECK_PARA;

  get_Sid_neuron(get_Soption_first, 0L, neuronList, "", &id, &neuronListStatus);
  while(neuronListStatus.all == status_Sok)
  {
    CHECK_ID(id);

    for(i = 0; i < neuronStat[id]->nrConnections; i++)
    {
      /* help compiler to perform CSE */
      connectionStat_t *c = &neuronStat[id]->connectionStat[i];

      if(c->pruned)
        continue;
      switch(neuronStat[id]->newActivation)
      {
        case threshold:
        case linear:
          pruneSwing = neuronStat[id]->angle * fabs(c->maxProd - c->minProd);
          break;
        case hardlimiter:
        case sigmoid:
          pruneSwing = fabs(c->maxProd - c->minProd);
```



```
          break;
        default:
          MyError("Internal neuronStat administration error, unkown newActivation: %d\n"
, neuronStat[id]->newActivation);
      }
      pruneSwing = fabs(c->maxProd - c->minProd);
      if(pruneSwing <= SWING_PRUNING_LEVEL)
      {
        RemoveConnection(neuronStat, neuronList, id, c->id);
        nrPruned++;
      }
    }
    get_Sid_neuron(get_Soption_larger, id, neuronList, "", &id, &neuronListStatus);
  }
  return nrPruned;
}


int SimplifyWeights(neuronStat_t **neuronStat, group_Sid_t neuronList)
/**
 ** IN: neuronList = InterAct neuron list, e.g. hidden_Slist
 **
 ** IN/OUT: neuronStat = pointer to neuronStat neuron administration
 **
 ** RET: number of connections simplified, -1 if error
 **
 ** DESCR:
 ** The bit-pruning algorithm. Determine number of required bits for the connections
 ** of the neurons registered in neuronList.
 **/
{
  status_St      status, neuronListStatus;
  neuron_Sid_t   id;
  neuron_Skind_t neuronKind;
  int            nrSimplified = 0;
  long           i;

  CHECK_PARA;

  get_Sid_neuron(get_Soption_first, 0L, neuronList, "", &id, &neuronListStatus);
  while(neuronListStatus.all == status_Sok)
  {
    CHECK_ID(id);

    for(i = 0; i < neuronStat[id]->nrConnections; i++)
    {
      /* help compiler to perform CSE */
      connectionStat_t *c = &neuronStat[id]->connectionStat[i];

      if(c->pruned)
        continue;
      if(fabs(c->weight) < 1.0)
        c->integerSize = 1;
      else
        c->integerSize = 1 + (int)ceil(log(fabs(c->weight)) / log(2.0));
      c->fractionSize = 0;
    }
    get_Sid_neuron(get_Soption_larger, id, neuronList, "", &id, &neuronListStatus);
  }
  return nrSimplified;
}


static void AdjustWeights(neuronStat_t **neuronStat, long id, group_Sid_t neuronList)
/**
 ** IN: id = id of neuron to be processed
 **     neuronList = Interact neuron list e.g. hidden_Slist
 **
 ** IN/OUT: neuronStat = pointer to neuronStat neuron administration
```

```
 **
 ** DESCR:
 ** Routine incorporates the calculated angle into the connection
 ** strenths of incoming weights. This simplifies pruning and code
 ** generation. This routine will only be used in case of a threshold
 ** or linear activation function replacement.
 **/
{
  status_St status;
  long i;

  if(neuronStat == NULL)
    MyError("neuronStat has not been initialized\n");
  if(neuronStat[id] == NULL)
    MyError("Interact administration error, unregistered neuronId: %ld\n", id);
  if(neuronStat[id]->angle == 0.0f)
    MyError("angle can't be zero, redundancy index classification error\n");

  for(i = 0; i < neuronStat[id]->nrConnections; i++)
  {
    /* help compiler to perform CSE */
    connectionStat_t *c = &neuronStat[id]->connectionStat[i];

    if(c->pruned)
      continue;
    /* change connection strenth, notify InterAct */
    c->weight *= neuronStat[id]->angle;
    set_Sweight_random(c->id, id, neuronList, neuronList, site_Sdefault_site, -FLT_MAX,
                       FLT_MAX, c->weight, 1.0f, rand_Sdummy, &status);
  }
}


static void threshDummy(void)
/**
 ** DESCR:
 ** A dummy, used by the change_Sthres function.
 **/
{
  /* empty */
}


/* these values are neuron parameters */
/* in this version the values are static network parameters */
#define SIGMOID_INDEX  5.0f
#define THRESHOLD_INDEX 700.0f
#define HARDLIMITER_INDEX 3000.0f

int SimplifyNeurons(neuronStat_t **neuronStat, group_Sid_t neuronList)
/**
 ** IN: neuronList = InterAct neuron list, e.g. hidden_Slist
 **
 ** IN/OUT: neuronStat = pointer to neuronStat neuron administration
 **
 ** RET: number of neurons (activation functions) simplified, -1 if error
 **
 ** DESCR:
 ** Routines simplifies the activation function of the neurons registered in neuronList.
 ** The selection process is based on a value called a redundancy index. After
 ** simplification several optimizations are performed concerning the angle and biases.
 **/
{
  status_St       status, neuronListStatus;
  float           biasUpdate;
  neuronStat_t    *neuron;
  neuron_Sid_t    id;
  neuron_Skind_t neuronKind;
  neuron_Sid_t    outputConnections[INTERACT_MAX_OUTPUTS];
```



```
  long            nrOutputConnections;
  int             nrSimplified = 0;
  long            i;

  CHECK_PARA;

  get_Sid_neuron(get_Soption_first, 0L, neuronList, "", &id, &neuronListStatus);
  while(neuronListStatus.all == status_Sok)
  {
    CHECK_ID(id);

    neuron = neuronStat[id];

    /* classification process */
    if(neuron->redundancyIndex < 0.0f)
    {
      fprintf(stderr, "WARNING: negative redundancy index, neuron: %ld  red.index: %f\n"
              id, neuron->redundancyIndex);
      neuron->redundancyIndex = fabs(neuron->redundancyIndex);
    }
    if(neuron->redundancyIndex < SIGMOID_INDEX)
      neuron->newActivation = sigmoid;
    else
    {
      nrSimplified++;

      /* LSQ */
      if(neuron->nrLSQPatterns > 0) /* otherwise neuron will be pruned */
      {
        neuron->angle = (neuron->nrLSQPatterns * neuron->LSQSumSumOut -
                         neuron->LSQSumSum * neuron->LSQSumOut) /
                        (neuron->nrLSQPatterns * neuron->LSQSumSquaredSum -
                         neuron->LSQSumSum * neuron->LSQSumSum);
        biasUpdate = neuron->LSQAvgOut / neuron->angle - neuron->LSQAvgSum;
        neuron->bias = neuron->oldBias + biasUpdate;
      }
      if(neuron->redundancyIndex < THRESHOLD_INDEX)
      {
        AdjustWeights(neuronStat, id, neuronList);
        neuron->bias *= neuron->angle;
        set_Sbias_random(id, neuronList, -FLT_MAX, FLT_MAX, neuron->bias, 1.0,
                         rand_Sdummy, &status);
        if(neuron->minOut > LSQ_MIN && neuron->maxOut < LSQ_MAX)
        {
          neuron->newActivation = linear;
          change_Sthres(id, neuronList, linear_transfer_St, 1.0, 1.0, 0.0, 1.0, 1.0,
                        threshDummy, &status);
        }
        else
        {
          neuron->newActivation = threshold;
          change_Sthres(id, neuronList, thrdet_sum_St, 1.0, 1.0, 0.0, 1.0,
                        neuron->angle, threshDummy, &status);
        }
      }
      else
      { /*
        if(neuron->redundancyIndex < HARDLIMITER_INDEX)
        {
          neuron->newActivation = threshold;
        }
        else
        {
        */
          neuron_Sid_t connectId;
          long j;

          neuron->newActivation = pruned;
```

```
          /* adjust bias of connected neurons */
          get_Sneuron_outputs(id, outputConnections, &nrOutputConnections, &status);
          for(i = 0; i < nrOutputConnections; i++)
          {
            connectId = outputConnections[i];

            if(neuronStat[connectId] == NULL)
              MyError("Interact administration error, unregistered neuronId: %ld\n",
                      connectId);
            /* linear search in private administration */
            for(j = 0; j < neuronStat[connectId]->nrConnections &&
                         neuronStat[connectId]->connectionStat[j].id != id; j++)
              ;
            if(j == neuronStat[connectId]->nrConnections)
              MyError("unregistered connection\n");
            neuronStat[connectId]->bias += neuronStat[id]->avgOut * neuronStat[connectId
]->connectionStat[j].weight;
            neuronStat[connectId]->connectionStat[j].pruned = TRUE;
            set_Sbias_random(connectId, neuronList, -FLT_MAX, FLT_MAX,
                             neuronStat[connectId]->bias, 1.0, rand_Sdummy, &status);
          }
          neuron_Sdelete(id, neuronList, &status);
    /*  } */
        }
      }
      get_Sid_neuron(get_Soption_larger, id, neuronList, "", &id, &neuronListStatus);
    }
    return nrSimplified;
}


#ifdef NO_PATTERN_LIST
  void NetworkAnalysis(float patternList[MAX_NOF_PATTERNS][MAX_NOF_INPUTS],
                       long patternListSize, long nrPatternInput, char *dFileName,
                       neuronStat_t **neuronStat)
/**
 ** IN: patternList = patternList used by OCR application
 **     patternListSize = nr. of patterns in patternList
 **     nrPatternInput = nr. of inputs to the network, needed for copy process
 **/
#else
  void NetworkAnalysis(pattern_Slist_id_t listId, char *dFileName,
                       neuronStat_t **neuronStat)
/**
 ** IN: listdId = InterAct neuron list e.g. hidden_Slist
 **/
#endif
/**
 ** IN: dFileName = string, ASCII name of the output filename and or pathname.
 **
 ** IN/OUT: neuronStat = pointer to neuronStat neuron administration
 **
 ** DESCR:
 ** Routine acts as a placeholder for the optimization routines in this module.
 ** An application  calls this routine when optimizations are required.
 ** If dFileName can not be opened an error is printed and the program is terminated.
 **/
{
  FILE   *dFile;
  time_t iTime;
  char   *aTime;
  int    totChanged = 0;

  if(dFileName != NULL)
  {
    if((dFile = fopen(dFileName, "w")) == NULL)
      MyError("%s could not be opened for write\n", dFileName);
  }
  else
```



```
    dFile = stderr;

  time(&iTime);
  aTime = ctime(&iTime);
  fprintf(dFile, "File created: %s\n\n", aTime);

  fprintf(stderr, "Starting network analysis\n");
  #ifdef NO_PATTERN_LIST
    GetNeuronStat(neuronStat, neuron_Slist, patternList, patternListSize,
                  nrPatternInput, dFile);
  #else
    GetNeuronStat(neuronStat, neuron_Slist, listId, dFile);
  #endif
  fprintf(stderr, "Analysis done\n");

  fprintf(stderr, "Starting neuron simplification process\n");
  totChanged += SimplifyNeurons(neuronStat, neuron_Slist);
  fprintf(stderr, "Starting weight simplification process\n");
  totChanged += SimplifyWeights(neuronStat, neuron_Slist);
/*
  fprintf(stderr, "Starting weight connection swing pruning process\n");
  totChanged +=  SwingConnectionPrunig(neuronStat, neuron_Slist);
*/
  fprintf(stderr, "Dumping administation to file: %s\n", dFileName);
  PrintNeuronStat(neuronStat, neuron_Slist, dFile);

  fprintf(stderr, "Done!\n\n");

  if(dFile != stderr)
    fclose(dFile);
}
```

```
/**
 **      FILE            : misc.h
 **      VERSION         : 1.0
 **      DATE            : 01-08-95
 **      AUTHOR          : H.Keegstra
 **                        csg675@cs.rug.nl
 **
 **      DESCRIPTION     : miscellaneous, some handy routines
 **                        used by many others
 **/

#ifndef _MISC_H_
#define _MISC_H

#include <stdio.h>
#include <stddef.h>

#define TRUE 1
#define FALSE 0

#ifndef __GNUC__
/**
 ** CAUTION: these are MACROS -> parameters will be evaluated more
 **          than once!!
 **/
#define min(x,y) (((x)<(y))? (x) : (y))
#define max(x,y) (((x)>(y))? (x) : (y))
#define sqr(x) ((x)*(x))
#else
/* GCC offers some neat alternatives (watch the GCC comma shield) */
#define min(x,y) \
  (( typeof (x) __x = (x), __y = (y); \
    (__x < __y) ? __x : __y; ))
#define max(x,y) \
  (( typeof (x) __x = (x), __y = (y); \
    (__x > __y) ? __x : __y; ))
#define sqr(x) \
  (( typeof (x) __x = (x); \
    __x * __x; ))
#endif

void MyError(const char *format, ...);
/**
 ** General error printing routine also terminates program.
 **/

void *MyMalloc(size_t size);
/**
 ** Malloc, including some necessary size checks.
 **/

void *MyCalloc(size_t size);
/**
 ** Calloc, including some necessary size checks.
 **/

void EPrintf(const char *format, ...);
/**
 ** Printf to stderr.
 **/

void ECPrintf(int condition, const char *format, ...);
/**
 ** Printf to stderr if condition is true.
 **/

#endif
```

```
/**
 **      FILE            : misc.c
 **      VERSION         : 1.0
 **      DATE            : 01-08-95
 **      AUTHOR          : H.Keegstra
 **                        csg675@cs.rug.nl
 **
 **      DESCRIPTION     : miscellaneous, some handy routines
 **                        used by many others
 **/

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <stddef.h>
#include "misc.h"


void MyError(const char *format, ...)
/**
 ** IN: format = the things to be printed
 **
 ** DESCR:
 ** General error printing routine also terminates program.
 **/
{
  va_list arg;

  va_start(arg, format);
  fprintf(stderr, "ERROR:");
  vfprintf(stderr, format, arg);
  fprintf(stderr, "\n");
  fflush(stderr);
  va_end(arg);
  exit(1);
}


void *MyMalloc(size_t size)
/**
 ** IN: size = the size in bytes of the block to be allocated
 **
 ** DESCR:
 ** Malloc, including some necessary size checks.
 **/
{
  void *p;

  if((p = malloc(size)) == NULL)
    MyError("out of memory\n");
  return p;
}


void *MyCalloc(size_t size)
/**
 ** IN: size = the size in bytes of the block to be allocated
 **
 ** DESCR:
 ** Calloc, including some necessary size checks.
 **/
{
  void *p;

  if((p = calloc(1, size)) == NULL)
    MyError("out of memory\n");
  return p;
}
```



```
void EPrintf(const char *format, ...)
/**
 ** IN: format = the things to be printed
 **
 ** DESCR:
 ** Printf to stderr.
 **/
{
  va_list arg;

  va_start(arg, format);
  vfprintf(stderr, format, arg);
  fflush(stderr);
  va_end(arg);
}


void ECPrintf(int condition, const char *format, ...)
/**
 ** IN: condition = control input, true means print format, false do nothing
 **     format = the things to be printed
 **
 ** DESCR:
 ** Printf to stderr if condition is true.
 **/
{
  va_list arg;

  if(condition)
  {
    va_start(arg, format);
    vfprintf(stderr, format, arg);
    fflush(stderr);
    va_end(arg);
  }
}
```