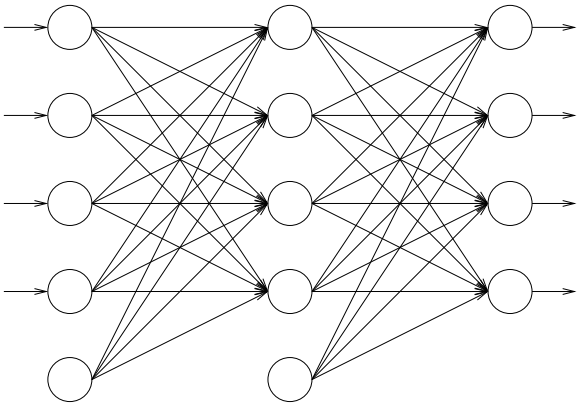
Implementation of a Fast Artificial Neural Network Library (fann)

Steffen Nissen
lukesky@diku.dk

October 31, 2003

Department of Computer Science
University of Copenhagen (DIKU)



Abstract

This report describes the implementation of a fast artificial neural network library in ANSI C called fann. The library implements multilayer feedforward networks with support for both fully connected and sparse connected networks. Fann offers support for execution in fixed point arithmetic to allow for fast execution on systems with no floating point processor. To overcome the problems of integer overflow, the library calculates a position of the decimal point after training and guarantees that integer overflow can not occur with this decimal point.

The library is designed to be fast, versatile and easy to use. Several benchmarks have been executed to test the performance of the library. The results show that the fann library is significantly faster than other libraries on systems without a floating point processor, while the performance was comparable to other highly optimized libraries on systems with a floating point processor.

Keywords: ANN, artificial neural network, performance engineering, fixed point arithmetic, ANSI C.

Preface

This report is written by Steffen Nissen as a graduate project on DIKU[1]. Associate Professor Klaus Hansen is connected to the project as supervisor.

The source code for this project can be found on the internet address: http://softman.dk/~lukesky/fann/. Furthermore the source code is also located in appendix B. The library will be released under the LGPL licence [FSF, 1999] accompanied by this report. It will be released as a SourceForge.net project [OSDN, 2003] on the internet address: http://SourceForge.net/projects/fann/, shortly after the completion of this report.

A CD-ROM accompanies this report, when delivered as a graduate project. The root of the CD-ROM consist of this report as a PDF file, a small README file and a directory called `fann`. This directory contains the entire CVS structure used while creating this library and report, I will now describe the important directories in this CVS:

doc Contains all the material used in the report, including some of the articles used for writing this report.

libraries Contains the ANN libraries and programs, which was used for the benchmarks.

src Contains the source for the fann library and the files generated during the benchmarks.

src/test Contains the test and benchmark programs.

src/datasets Contains the datasets used for the quality benchmarks.

[1] DIKU: Datalogisk Institut Københavns Universitet, Department of Computer Science University of Copenhagen

Contents

1 Introduction

This report focuses on the process of implementing a fast artificial neural network library. Many performance enhancement techniques will be used and documented, to allow the library to be as fast as possible.

The report is written in English to be able to reach a broader audience. It is my hope that people interested in implementing an application in the field of artificial intelligence and computerized learning, will be able to read this report as an introduction to programming with an artificial neural network library. And that people who have already written an application using neural networks, can use this report as a guide on how to increase the performance of their application using the fann library.

1.1 Motivation

In [Nissen et al., 2002] I participated in building and programming of an autonomous robot, based on an Compaq iPAQ with a camera attached to it. In [Nissen et al., 2003] I participated in rebuilding this robot and adding artificial neural networks (ANN) for use in the image processing. Unfortunately the ANN library that we used [Heller, 2002] was too slow and the image processing on the iPAQ was not efficient enough.

The iPAQ does not have a floating point processor and for this reason we had written a lot of the image processing using fixed point arithmetic. From this experience I have learned that rewriting code to fixed point arithmetic, makes a huge difference on the performance of programs running on the iPAQ.

This experience gave me the idea, that it might be useful to develop a fast ANN library with support for fixed point arithmetic. I did however not think that writing another project based on the robot would be a good idea, so I started speculating on what I wanted to do for my master thesis.

For my master thesis I would like to make an autonomous agent, which learns from experience. To allow the agent to operate in a virtual environment, I have chosen the field of computer games. The agent should be able to function as a virtual player in a computer game and learn while doing so. Many different games could be used for this purpose, but I am leaning towards Quake III Arena [IDS, 2000], because it already has artificial agents called game bots [van Waveren, 2001]. Part of the code for these bots are public domain and it should be possible to write a new game bot on the basis of this code.

One way of building a learning game bot is to use reinforcement learning algorithms [Kaelbling et al., 1996]. Reinforcement learning can benefit from having an ANN core, which makes it useful to have a fast ANN library that can be modified for use as a reinforcement learning core. Quake III Arena is written in ANSI C and for this reason the ANN library used as the reinforcement learning core should also be written in ANSI C.

This research lead back to the need for a fast ANN library. So I decided to implement a fast ANN library which could be not only be used by me, but also by other developers.

1.2 Requirements For the Library

The primary aim of this project, is to implement a fast ANN library. However, a fast library is an ambiguous term and although it is the primary aim, it is not the only aim. Therefore the aim will be discussed and specified further.

The primary aim is for the library to be fast at executing the ANN, while training the ANN is not as time critical. The execution of the library should be fast on both

systems which has a floating point processor and on systems which does not have a floating point processor.

Although I have some real needs for this library, there really is no point in developing a library which is not used by other developers. For this reason the library should be easy to use, versatile, well documented and portable.

Since the library should be used in Quake III Arena, it should be written in ANSI C. This also supports the requirement for portability, since ANSI C libraries can be used from a wide variety of programming languages and operating systems.

1.3 The Structure of This Report

This report have two main purposes:

- A report documenting the analysis, design and implementation of a library, where speed is of importance and benchmarking the library to document the performance and to discover, which optimizations have significant influence on performance.
- A complete user's guide to the fann library, making it possible for people with no prior knowledge of ANNs to read this report and start using the library and making it possible for people with extensive knowledge of ANNs to read parts of this report and discover, how they could benefit from using the fann library.

The report is divided into five main sections:

Neural Network Theory Describes the neural network theory needed in order to understand the rest of the report.

Analysis Analyzes what the library should be able to do and which methods should be used in order to reach this goal.

Design and Implementation Lowlevel API, architectural and algorithmic design.

User's Guide A guide to using the library. This guide is partly written for people wanting to use the library and partly written to document the versatility and userfriendliness of the library.

Benchmarks Documentation of the quality and performance of the library.

2 Neural Network Theory

This section will briefly explain the theory of neural networks (hereafter known as NN) and artificial neural networks (hereafter known as ANN). For a more in depth explanation of these concepts please consult the literature; [Hassoun, 1995] has good coverage of most concepts of ANN and [Hertz et al., 1991] describes the mathematics of ANN very thoroughly, while [Anderson, 1995] has a more psychological and physiological approach to NN and ANN. For the pragmatic I could recommend [Tettamanzi and Tomassini, 2001], which has a short and easily understandable introduction to NN and ANN.

2.1 Neural Networks

The human brain is a highly complicated machine capable of solving very complex problems. Although we have a good understanding of some of the basic operations that drive the brain, we are still far from understanding everything there is to know about the brain.

In order to understand ANN, you will need to have a basic knowledge of how the internals of the brain work. The brain is part of the central nervous system and consists of a very large NN. The NN is actually quite complicated, but I will only include the details needed to understand ANN, in order to simplify the explanation.

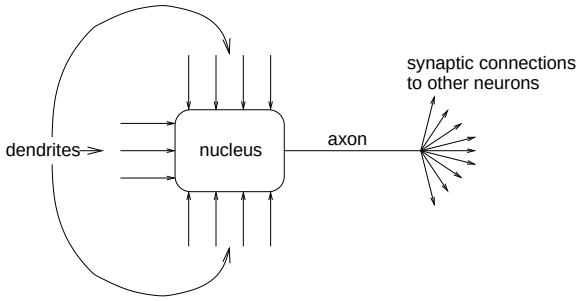


Figure 1: Simplified neuron.

The NN is a network consisting of connected neurons. The center of the neuron is called the nucleus. The nucleus is connected to other nucleuses by means of the dendrites and the axon. This connection is called a synaptic connection.

The neuron can fire electric pulses through its synaptic connections, which is received at the dendrites of other neurons. Figure 1 shows how a simplified neuron looks like.

When a neuron receives enough electric pulses through its dendrites, it activates and fires a pulse through its axon, which is then received by other neurons. In this way information can propagate through the NN. The synaptic connections change throughout the lifetime of a neuron and the amount of incoming pulses needed to activate a neuron (the threshold) also change. This behavior allows the NN to learn.

The human brain consists of around 10^{11} neurons which are highly interconnected with around 10^{15} connections [Tettamanzi and Tomassini, 2001]. These neurons activates in parallel as an effect to internal and external sources. The brain is connected to the rest of the nervous system, which allows it to receive information by means of the five senses and also allows it to control the muscles.

2.2 Artificial Neural Networks

It is not possible (at the moment) to make an artificial brain, but it is possible to make simplified artificial neurons and artificial neural networks. These ANNs can be made in many different ways and can try to mimic the brain in many different ways.

ANNs are not intelligent, but they are good for recognizing patterns and making simple rules for complex problems. They also have excellent training capabilities which is why they are often used in artificial intelligence research.

ANNs are good at generalizing from a set of training data. E.g. this means an ANN given data about a set of animals connected to a fact telling if they are mammals or not, is able to predict whether an animal outside the original set is a mammal from its data. This is a very desirable feature of ANNs, because you do not need to know the characteristics defining a mammal, the ANN will find out by itself.

2.2.1 The Artificial Neuron

A single artificial neuron can be implemented in many different ways. The general mathematic definition is as showed in equation 2.1.

$$y(x) = g\left(\sum_{i=0}^{n} w_i x_i\right) \tag{2.1}$$

x is a neuron with n input dendrites ($x_0 \ \ldots \ x_n$) and one output axon $y(x)$ and where ($w_0 \ \ldots \ w_n$) are weights determining how much the inputs should be weighted.

g is an activation function that weights how powerful the output (if any) should be from the neuron, based on the sum of the input. If the artificial neuron should mimic a real neuron, the activation function g should be a simple threshold function returning 0 or 1. This is however, not the way artificial neurons are usually implemented. For many different reasons it is smarter to have a smooth (preferably differentiable) activation function. The output from the activation function is either between 0 and 1, or between -1 and 1, depending on which activation function is used. This is not entirely true, since e.g. the identity function, which is also sometimes used as activation function, does not have these limitations, but most other activation functions uses these limitations. The inputs and the weights are not restricted in the same way and can in principle be between $-\infty$ and $+\infty$, but they are very often small values centered around zero. The artificial neuron is also illustrated in figure 2.

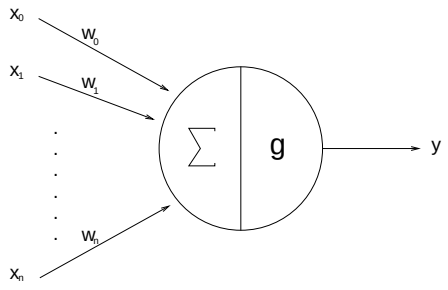


Figure 2: An artificial neuron.

In the figure of the real neuron (figure 1), the weights are not illustrated, but they are implicitly given by the number of pulses a neuron sends out, the strength of the pulses and how closely connected the neurons are.

As mentioned earlier there are many different activation functions, some of the most commonly used are threshold (2.2), sigmoid (2.3) and hyperbolic tangent (2.4).

$$g(x) = \begin{cases} 1 & if \quad x + t > 0 \\ 0 & if \quad x + t \leq 0 \end{cases} \tag{2.2}$$

$$g(x) = \frac{1}{1 + e^{-2s(x+t)}} \tag{2.3}$$

$$\begin{aligned} g(x) &= \tanh(s(x+t)) = \frac{\sinh(s(x+t))}{\cosh(s(x+t))} \\ &= \frac{e^{s(x+t)} - e^{-s(x+t)}}{e^{s(x+t)} + e^{-s(x+t)}} = \frac{e^{2(s(x+t))} - 1}{e^{2(s(x+t))} + 1} \end{aligned} \tag{2.4}$$

Where t is the value that pushes the center of the activation function away from zero and s is a steepness parameter. Sigmoid and hyperbolic tangent are both smooth differentiable functions, with very similar graphs, the only major difference is that hyperbolic tangent has output that ranges from -1 to 1 and sigmoid has output that ranges from 0 to 1. A graph of a sigmoid function is given in figure 3, to illustrate how the activation function look like.

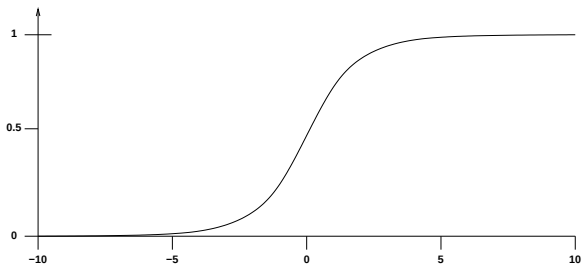


Figure 3: A graph of a sigmoid function with $s = 0.5$ and $t = 0$.

The t parameter in an artificial neuron can be seen as the amount of incoming pulses needed to activate a real neuron. This parameter, together with the weights, are the parameters adjusted when the neuron learns.

2.2.2 The Artificial Neural Network

The ANN library I have chosen to implement is a multilayer feedforward ANN, which is the most common kind of ANN. In a multilayer feedforward ANN, the neurons are ordered in layers, starting with an input layer and ending with an output layer. Between these two layers are a number of hidden layers. Connections in these kinds of network only go forward from one layer to the next. Many other kinds of ANNs exists, but I will not explain them further here. [Hassoun, 1995] describes several of these other kinds of ANNs.

Multilayer feedforward ANNs have two different phases: A training phase (sometimes also referred to as the learning phase) and an execution phase. In the training phase the ANN is trained to return a specific output when given a specific input, this is done by continuous training on a set of training data. In the execution phase the ANN returns outputs on the basis of inputs.

The way the execution of a feedforward ANN functions are the following: An input is presented to the input layer, the input is propagated through all the layers (using equation 2.1) until it reaches the output layer, where the output is returned. In a feedforward ANN an input can easily be propagated through the network and evaluated to an output. It is more difficult to compute a clear output from a network where connections are allowed in all directions (like in the brain), since

this will create loops. There are ways of dealing with these loops in recurrent networks, ([Hassoun, 1995] p. 271) describes how recurrent networks can be used to code time dependencies, but feedforward networks are usually a better choice for problems that are not time dependent.

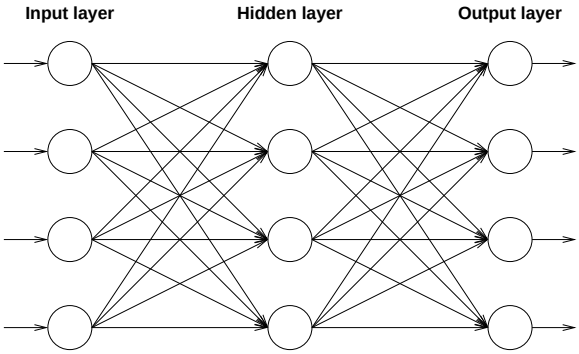


Figure 4: A fully connected multilayer feedforward network with one hidden layer.

Figure 4 shows a multilayer feedforward ANN where all the neurons in each layer are connected to all the neurons in the next layer. This is called a fully connected network and although ANNs do not need to be fully connected, they often are.

Two different kinds of parameters can be adjusted during the training of an ANN, the weights and the t value in the activation functions. This is impractical and it would be easier if only one of the parameters should be adjusted. To cope with this problem a bias neuron is invented. The bias neuron lies in one layer, is connected to all the neurons in the next layer, but none in the previous layer and it always emits 1. Since the bias neuron emits 1 the weights, connected to the bias neuron, are added directly to the combined sum of the other weights (equation 2.1), just like the t value in the activation functions. A modified equation for the neuron, where the weight for the bias neuron is represented as w_{n+1}, is shown in equation 2.5.

$$y(x) = g\left(w_{n+1} \sum_{i=0}^{n} w_i x_i\right) \tag{2.5}$$

Adding the bias neuron allows us to remove the t value from the activation function, only leaving the weights to be adjusted, when the ANN is being trained. A modified version of the sigmoid function is shown in equation 2.6.

$$g(x) = \frac{1}{1 + e^{-2sx}} \tag{2.6}$$

We cannot remove the t value without adding a bias neuron, since this would result in a zero output from the sum function if all inputs where zero, regardless of the values of the weights. Some ANN libraries do however remove the t value without adding bias neurons, counting on the subsequent layers to get the right results. An ANN with added bias neurons is shown in figure 5.

2.2.3 Running Time of Executing an ANN

When executing an ANN, equation 2.5 needs to be calculated for each neuron which is not an input or bias neuron. This means that we have to do one multiplication and one addition for each connection (including the connections from the bias neurons), besides that we also need to make one call to the activation function for each neuron that is not an input or bias neuron. This gives the following running time:

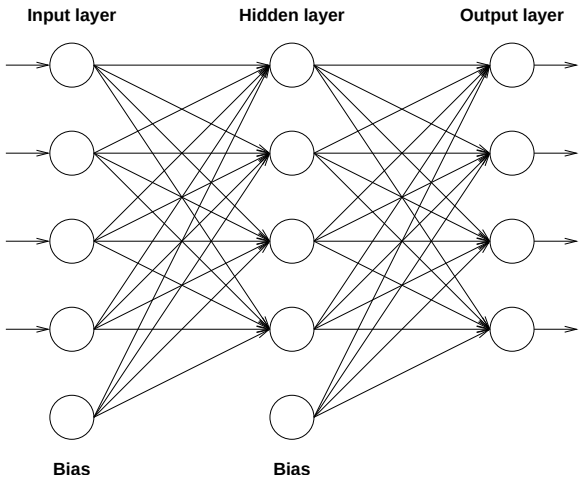


Figure 5: A fully connected multilayer feedforward network with one hidden layer and bias neurons.

$$T = cA + (n - n_i)G \tag{2.7}$$

Where c is the number of connections, n is the total number of neurons, n_i is the number of input and bias neurons, A is the cost of multiplying the weight with the input and adding it to the sum, G is the cost of the activation function and T is the total cost.

If the ANN is fully connected, l is the number of layers and n_l is the number of neurons in each layer (not counting the bias neuron), this equation can be rewritten to:

$$T = (l - 1)(n_l^2 + n_l)A + (l - 1)n_l G \tag{2.8}$$

This equation shows that the total cost is dominated by A in a fully connected ANN. This means that if we want to optimize the execution of a fully connected ANN, we need to optimize A and retrieval of the information needed to compute A.

2.3 Training an ANN

When training an ANN with a set of input and output data, we wish to adjust the weights in the ANN, to make the ANN give the same outputs as seen in the training data. On the other hand, we do not want to make the ANN too specific, making it give precise results for the training data, but incorrect results for all other data. When this happens, we say that the ANN has been over-fitted.

The training process can be seen as an optimization problem, where we wish to minimize the mean square error of the entire set of training data (for further information on the mean square error see section 5.3.3). This problem can be solved in many different ways, ranging from standard optimization heuristics like simulated annealing, through more special optimization techniques like genetic algorithms to specialized gradient descent algorithms like backpropagation.

The most used algorithm is the backpropagation algorithm (see section 2.3.1), but this algorithm has some limitations concerning, the extent of adjustment to the weights in each iteration. This problem has been solved in more advanced algorithms like RPROP [Riedmiller and Braun, 1993] and quickprop [Fahlman, 1988], but I will not elaborate further on these algorithms.

2.3.1 The Backpropagation Algorithm

The backpropagation algorithm works in much the same way as the name suggests: After propagating an input through the network, the error is calculated and the error is propagated back through the network while the weights are adjusted in order to make the error smaller. When I explain this algorithm, I will only explain it for fully connected ANNs, but the theory is the same for sparse connected ANNs.

Although we want to minimize the mean square error for all the training data, the most efficient way of doing this with the backpropagation algorithm, is to train on data sequentially one input at a time, instead of training on the combined data. However, this means that the order the data is given in is of importance, but it also provides a very efficient way of avoiding getting stuck in a local minima.

I will now explain the backpropagation algorithm, in sufficient details to allow an implementation from this explanation:

First the input is propagated through the ANN to the output. After this the error e_k on a single output neuron k can be calculated as:

$$e_k = d_k - y_k \tag{2.9}$$

Where y_k is the calculated output and d_k is the desired output of neuron k. This error value is used to calculate a δ_k value, which is again used for adjusting the weights. The δ_k value is calculated by:

$$\delta_k = e_k \; g'(y_k) \tag{2.10}$$

Where g' is the derived activation function. The need for calculating the derived activation function was why I expressed the need for a differentiable activation function in section 2.2.1.

When the δ_k value is calculated, we can calculate the δ_j values for preceding layers. The δ_j values of the previous layer is calculated from the δ_k values of this layer. By the following equation:

$$\delta_j = \eta \; g'(y_j) \sum_{k=0}^{K} \delta_k w_{jk} \tag{2.11}$$

Where K is the number of neurons in this layer and η is the learning rate parameter, which determines how much the weight should be adjusted. The more advanced gradient descent algorithms does not use a learning rate, but a set of more advanced parameters that makes a more qualified guess to how much the weight should be adjusted.

Using these δ values, the Δw values that the weights should be adjusted by, can be calculated by:

$$\Delta w_{jk} = \delta_j \; y_k \tag{2.12}$$

The Δw_{jk} value is used to adjust the weight w_{jk}, by $w_{jk} = w_{jk} + \Delta w_{jk}$ and the backpropagation algorithm moves on to the next input and adjusts the weights according to the output. This process goes on until a certain stop criteria is reached. The stop criteria is typically determined by measuring the mean square error of the training data while training with the data, when this mean square error reaches a certain limit, the training is stopped. More advanced stopping criteria involving both training and testing data are also used.

In this section I have briefly discussed the mathematics of the backpropagation algorithm, but since this report is mainly concerned with the implementation of ANN algorithms, I have left out details unnecessary for implementing the algorithm. I will refer to [Hassoun, 1995] and [Hertz et al., 1991] for more detailed explanation of the theory behind and the mathematics of this algorithm.

2.3.2 Running Time of Backpropagation

The backpropagation algorithm starts by executing the network, involving the amount of work described in section 2.2.3 in addition to the actual backpropagation.

If the ANN is fully connected, the running time of algorithms on the ANN is dominated by the operations executed for each connection (as with execution of an ANN in section 2.2.3).

The backpropagation is dominated by the calculation of the δ_j and the adjustment of w_{jk}, since these are the only calculations that are executed for each connection. The calculations executed for each connection when calculating δ_j is one multiplication and one addition. When adjusting w_{jk} it is also one multiplication and one addition. This means that the total running time is dominated by two multiplications and two additions (three if you also count the addition and multiplication used in the forward propagation) per connection. This is only a small amount of work for each connection, which gives a clue to how important it is, for the data needed in these operations to be easily accessible.

3 Analysis

In this section I will analyze the requirements of the library and the methods needed to meet these requirements. A major requirement is the demand for speed and for this reason I will analyze numerous methods for optimizing speed.

3.1 Usage Analysis

The fann library is not intended to be used by me alone, it is intended to be used by many people. For the library to be a success in this area, it should be fast, easy to use and versatile enough to allow for it to be used in many different situations. A common mistake in software design is to keep adding more and more functionality without thinking about user friendliness. This added functionality can make the system difficult to comprehend for a user, which is why I will try to hide the inner architecture for the user and only expose functions needed by the user. ANNs have the appealing feature, that if implemented correctly they can be used by people, who have only very little understanding of the theory behind them.

I have already mentioned, that the kind of network I will implement is a feed-forward network and that it should be fast. What I have not mentioned is which functionalities I would like to implement in the library and which I would not like to implement.

The library should be easy to use and easy to train. For this reason, I would like to be able to train a network directly from training data stored in a file (with one function call). I would also like to be able to save and load all information about an ANN to and from a file. The most basic ANN operations should be very easy to use. The basic operations are: creating a network with reasonable defaults, training a network and executing a network.

I would like to make the library easy to alter in the future, which is why it should be possible to make sparse connected ANNs. I would also like to implement at least one of the possible activation functions, but still allowing more to be added later.

It should be possible to alter all the parameters mentioned in the theory (section 2) at runtime. The parameters are: learning rate, activation function, the steepness value of the activation function and the values of the initial weights.

It should not be possible to hand design a network topology, but it should be possible to create a network and decide how many layers there should be, how many neurons there should be in each of these layers and how dense the network should be. I will get back to the density in section 4.3, but generally what I want, is to be able to create a network and say that e.g. only half of the possible connections should be connected.

Some ANN packages have GUIs for viewing information about an ANN, but I do not think that it is the primary goal of an ANN library and for this reason I do not want to implement this. Due to the flexible nature of the library that I will implement (a network can be saved to and loaded from a file), it would be possible to make a stand-alone program that could implement these kinds of features.

3.2 Fixed Point Analysis

As mentioned in section 1.1, the ANN should be able to run with fixed point numbers. This however raises a lot of questions and not all of them have easy answers.

The first big question is how much of the functionality implemented in the floating point library should also be implemented in the fixed point library. The obvious answer to this question would be all of the functionality, this however raises a new question: What should be done when overflow occurs?

The hardware integer overflow exception is usually masked by the operating system or the compiler[2]. This implies that the only real alternatives are to check for overflow on each calculation or not to check for overflow at all. To check for overflow on each calculation would be too costly and would void the whole idea of using fixed point arithmetic for greater speed. On the other hand not to check at all would create overflow and unpredictable results in consequence.

This is an annoying problem, especially because you have no real control over the values of the weights. Usually in fixed point arithmetic you either give a guarantee, that there will never be integer overflow or you make a simple check, that can see if an overflow has occurred during a series of calculations. I can not find any simple check, that can guarantee that there has not been an overflow for a series of operations, but what I can do, is to guarantee that an overflow will never occur.

In order to make a guarantee that an overflow will never occur, I will have to reevaluate the amount of functionality which should be implemented in the fixed point library. Since fixed point arithmetic is mostly geared towards portable computers, it is safe to assume that there will be a standard PC available for training the ANN. This means that the training part of the ANN does not need to be implemented in fixed point. Another observation is that after the ANN is fully trained, the ANN never changes and it is therefore possible to make one check, after the training has finished, that will guarantee that an overflow will never occur.

These observations about the problem of fixed point arithmetic, give rise to several different implementation strategies. In section 4.4 I will choose an appropriate strategy and prove that there can be no overflow using this strategy.

3.3 Performance Analysis

The primary aim of the library is to be as fast as possible during training and execution of the ANN. To reach this aim, I will consider several kinds of optimization techniques. The techniques are partly inspired by the performance engineering course at DIKU and the rules defined in [Bentley, 1982] and partly by general common sense. The optimization techniques that I will consider are the following:

- Algorithmic optimization
- Architectural optimization
- Cache optimization
- Common subexpression elimination
- In-lining of code
- Specializations for fully connected ANNs
- Loop unrolling
- Table lookup

The cache optimizations are the most efficient, as can be seen in the benchmarks (section 6).

3.3.1 Algorithmic Optimization

When optimizing a piece of software, you will often find the most efficient improvements, in the algorithms used for the software. If you could change the running time of a piece of software, from $\Theta(n^2)$ to $\Theta(n)$ then this optimization would almost certainly be better than all other optimizations you could think of.

[2] In gcc it is possible to get signals when a overflow occurs with `-ftrapv`, but unfortunately some optimized pointer arithmetic in gcc makes integer overflow and breaks this functionality (see `http://gcc.gnu.org/bugzilla/show_bug.cgi?id=1823`).

The backpropagation algorithm will have to visit all connections, this cannot be changed and it is therefore not possible to change the running time of the backpropagation algorithm. However, as described in section 2.3, other more advanced algorithms exists which could get better results than the backpropagation algorithm. These algorithms do not execute faster than the backpropagation algorithm, but they adjust the weights more precise, making them reach a result faster.

I have chosen to implement the backpropagation algorithm, because it is simple and effective enough in most cases. This decision means that I have knowingly not implemented an important optimization for the training algorithm, which implies that there is not much use in spending too much time on the other optimization strategies, because a highly tuned backpropagation algorithm will still be slower than an untuned RPROP algorithm. In spite of that, a basic level of optimization is still a desirable feature in the implementation of the backpropagation algorithm.

In conclusion; not much is done about the algorithms (although something could be done about the training), which means that the running time is still $\Theta(n)$, where n is the number of connections. However, there is still room for optimization of the overhead involved in executing the actual calculations.

3.3.2 Architectural Optimization

There are many ways of building the architecture (data structures) for a neural network. The object oriented approach would be to make everything an object and there are actually good abstract concepts like neurons, synapses etc. which would make for a great class hierarchy. In Jet's Neural Library [Heller, 2002] such an approach has been chosen, with all the advantages and disadvantages of this choice. There are several major disadvantage of this approach:

- Data itself are not located closely together and cache performance is very bad.
- Algorithms like executing the network has code located in several different classes, which makes the code hard to optimize and adds an overhead on several key functions.
- It is difficult to make tight inner loops.

These are obviously problems that could be fixed, while still using the object oriented approach, but the object oriented approach makes it difficult to do so.

A good architecture for a neural network should not take up too much space and should not include too deep a level of objects. On the other hand some level of object abstraction is highly desired. Perhaps a three level hierarchy would be acceptable, with the outer level consisting of the entire ANN, the next level consisting of the individual layers and the last level consisting of the single neurons and connections.

A good architecture will also allow for easy access to information like total number of neurons etc.

3.3.3 Cache Optimization

If a good data architecture is in place much of the work for the cache optimization is already done. But some work still needs to be done in improving the architecture and making sure that the algorithms themselves are cache aware.

The architecture should assure that data could be accessed sequentially for good cache performance. A good example of this is the weights, which should be accessed sequentially when executing the network. For this reasons the weights should be aligned in memory in one long array, which could be accessed sequentially.

The algorithms themselves should obviously use this optimized architecture and access the data sequentially. The algorithms should also assure that all the code,

that they execute, are located at the same place to utilize the code cache to an optimum.

3.3.4 Common Subexpression Elimination

Many expressions are calculated several times in standard neural network algorithms. Although a compiler can do common subexpression elimination, it is often a good idea to calculate expressions only once and store them in local variables. A person can often do a better job at this, because a person can predict side effects and aliasing[3], which the compiler can not predict.

This is especially a good idea for the stop criteria of a loop, because this calculation is made in each run of the loop. If some of this calculation could be made only once, this would make for a good performance increase. Also variables from the ANN which is used in central loops could be prefetched to a local variable to avoid overhead of fetching the variable from memory each time.

The central algorithms should be hand optimized to evaluate all common subexpressions at an early state, while the not so central algorithms should let the compiler take care of this optimization.

3.3.5 In-lining of Code

All code which is evaluated more than once in either execution or training of the ANN, should be in-lined in the algorithm. This will avoid unnecessary overhead for function calls and allow the compiler to do optimizations across the function call.

The in-lining can be done by either writing the code directly in the algorithm, by using in-line functions or macros.

3.3.6 Specializations for Fully Connected ANNs

In fully connected ANNs we already know the connections between two layers. If we assure that the weights for fully connected ANNs are always located at the same place, we can implement algorithms which benefit from this information. This information can be used to access the weight independently of the information stored about connections.

Such an optimization benefits the performance in two ways: First of all, we can completely eliminate the need for using the memory, which store information about connections. Secondly, we can access the weights in one step less (one pointer reference instead of two).

3.3.7 Loop Unrolling

Unrolling loops can often be done more efficient by hand than by a compiler. This is partly because the compiler has to deal with aliasing, where a programmer can see that aliasing will not happen and make faster code.

A short example of this is:

```
a[0] = b[0];
a[0] += b[1];
a[0] += b[2];
```

Which could be rewritten by a programmer to the following (if the programmer was sure that `a` and `b` did not share data):

[3] C and C++ automaticly thinks that data reached from two different pointers could be the same. This makes for safe but slow code (FORTRAN assumes the opposite, which makes for fast unsafe code).

```
a[0] = b[0] + b[1] + b[2];
```

The compiler can not do this, because it can not be sure that `b[1]` and `b[2]` are not sharing the same memory as `a[0]` and is therefore altered by `a[0] = b[0];`.

3.3.8 Table Lookup

As seen in figure 3, the activation functions are often very close to zero for small values and close to one for large values. This leaves a relatively small span where the output is not zero or one. This span can be represented as a lookup table, with a reasonable resolution. It is hard to tell whether this lookup table will be faster than actually calculating the activation function, but it is worth a try.

4 Design and Implementation

In section 3 I have analyzed what the library should be able to do and which methods should be used to reach this objective. In this section I will use these considerations to give concrete suggestions as to how the design and programming of the library should be constructed. I will also describe how I have implemented some of these suggestions and why I have not implemented others. If nothing else is stated, all suggestions from both analysis and design have been implemented.

4.1 API Design

Much of the API have already been sketched in the "Usage Analysis" (section 3.1), so I will only give a few more details in this section and leave the actual description of the API to the "User's Guide" (section 5).

Since the library should be written in ANSI C, the API needs to be a function based API, but there can still be an object oriented thought behind the API.

I will use an ANN structure, which can be allocated by a constructor and deallocated by a destructor. This structure should be given as the first argument to all functions which operates on the ANN, to mimic an object oriented approach.

The ANN should have three different methods of storing the internal weights: `float`, `double` and `int`. Where `float` and `double` are standard floating point representations and `int` is the fixed point representation. In order to give the compiler the best possible opportunity to optimize the code, this distinction should be made at compile-time. This will produce several different libraries and require that the person using the library should include a header file which is specific to the method chosen. Although there is this distinction between which header file is include, it should still be easy to write code which could compile with all three header files. For this purpose I have invented a `fann_type`, which is defined in the three header files as `float`, `double` and `int` respectively.

It should be possible to save the network in standard floating point representation and in fixed point representation (more on this in section 4.4).

Furthermore there should be a structure which could hold training data. This structure should like the net itself be loadable from a file. The structure of the file should be fairly simple, making it easy to export a file in this format from another program.

I will leave the rest of the API details to section 5 "User's Guide".

4.2 Architectural Design

In section 3.3.2 "Architectural Optimization" I have outlined how I will create the general architectural design. In this section I will specify more precisely how the design should be.

The data structures should be structured in three levels: A level containing the whole ANN, a level containing the layers and a level containing the neurons and connections.

With this three level structure, I will suggest two different implementations. The first is centered around the connections and the second is centered around the neurons.

4.2.1 Connection Centered Architecture

In a structure where the connections are the central structure, the three levels would look like this:

1. `fann` The ANN with references to the connection layers.

2. `fann_connection_layer` The connection layers with references to the connections.
3. `fann_connection` The connections with a weight and two pointers to the two connected neurons.

Where the connection layers represent all the connections between two layers and the neurons themselves are only basic `fann_type`'s. The main advantage of this structure is that one simple loop can run through all the connections between two layers. If these connections are allocated in one long array, the array can be processed completely sequentially. The neurons themselves are only basic `fann_type`'s and since there are far less neurons than there are connections they do not take up that much space. This architecture is shown on figure 6.

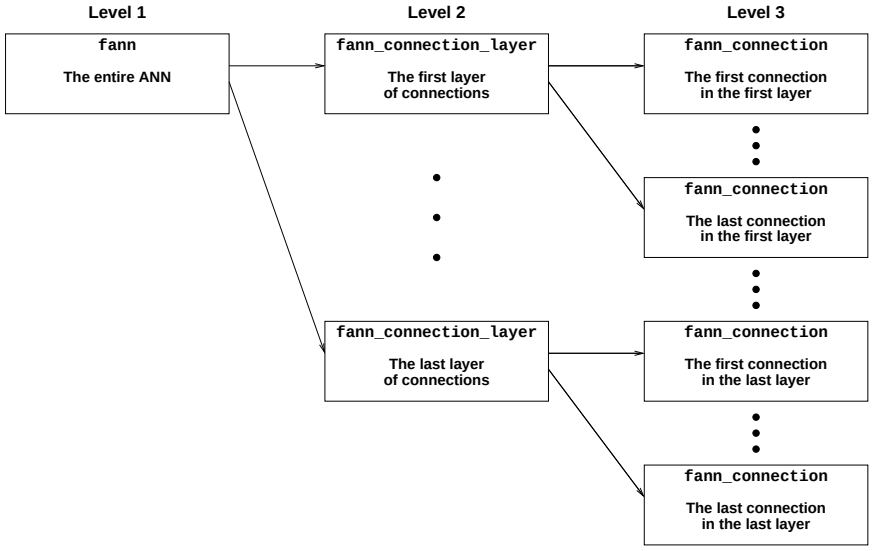


Figure 6: The three levels for the connection centered architecture. At level 3, the `fann_connection` structures consists of a weight and a pointer to each of the two connected neurons.

This was the first architecture which I implemented, but after testing it against Lightweight Neural Network version 0.3 [van Rossum, 2003], I noticed that this library was more than twice as fast as mine.

The question is now, why was the connection centered architecture not as fast as it should have been? I think the main reasons was that the connections, which is the most represented structure in ANNs, was implemented by a struct with two pointers and a weight. This took up more space than needed which meant poor cache performance. It also meant that the inner loop should constantly dereference pointers, which the compiler had no idea where would end up, although they where actually accessing the neurons in a sequential order.

4.2.2 Neuron Centered Architecture

After an inspection of the architecture used in the Lightweight Neural Network and evaluation of what went wrong in the connection centered architecture, I designed the neuron centered architecture, where the three levels looks like this:

1. `fann` The ANN with references to the layers.
2. `fann_layer` The layers with references to the neurons.

3. `fann_neuron` The neurons with references to the connected neurons in the previous layer and the weights for the connections.

In this architecture the `fann` structure has a pointer to the first layer and a pointer to the last layer[4]. Furthermore, it contains all the parameters of the network, like learning rate etc. The individual layers consists of a pointer to the first neuron in the layer and a pointer to the last neuron in the layer. This architecture is shown on figure 7.

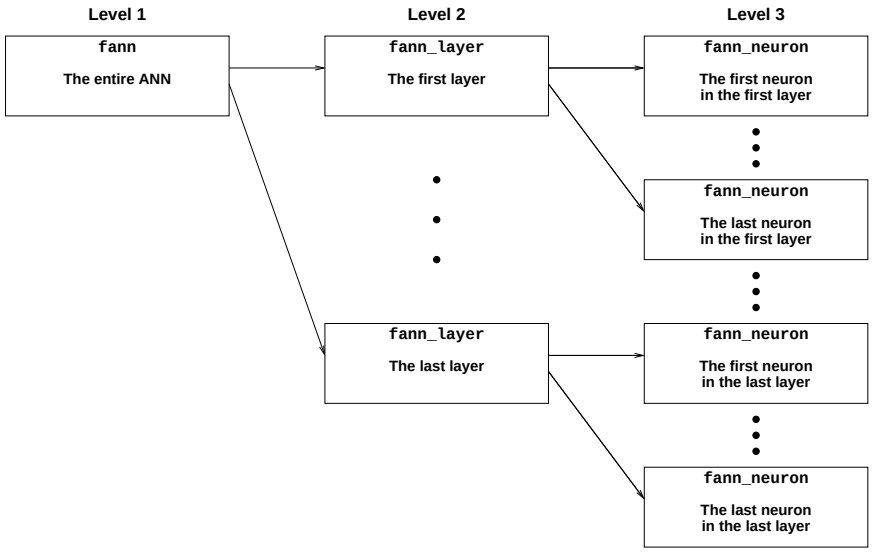


Figure 7: The three levels for the neuron centered architecture. At level 3, the `fann_neuron` structures consists of a neuron value, a value for the number of neurons from the previous layer, connected to the neuron and two pointers. The first pointer points at the first weight for the incoming connections and the second pointer points to the first incoming connection (see figure 8).

Figure 8 illustrates how the architecture is connected at the third level. The neurons, the weights and the connections are allocated in three long arrays. A single neuron N consists of a neuron value, two pointers and a value for the number of neurons connected to N in the previous layer. The first pointer points at the position in the weight array, which contain the first weight for the connections to N. Likewise the second pointer points to the first connection to N in the connection array. A connection is simply a pointer to a neuron, in the previous layer, which is connected to N.

This architecture is more cache friendly, than the connection centered architecture. When calculating the input to a neuron, the weights, the connections and the input neurons are all processed sequentially. The value of the neuron itself is allocated in a local variable, which makes for much faster access. Furthermore, the weights and the connections are all processed completely sequentially, throughout the whole algorithm. When the ANN is fully connected, the array with the connected neurons is obsolete and the algorithms can use this to increase speed.

[4]Actually this is a past the end pointer, which points to the layer one position past the last layer. But it is easier to think of it as a pointer to the last layer.

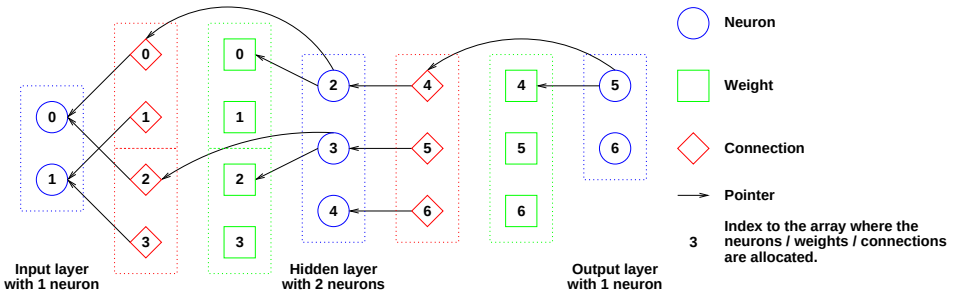


Figure 8: Architecture of the fann library at the third level, illustrated by an ANN with 1 input neuron, two neurons in a hidden layer and one output neuron. Bias neurons are also illustrated, including a bias neuron in the output layer which is not used, but still included to make algorithms more simple. Three things should be noticed on this figure, the first is that all pointers go backwards which is more optimal when executing the ANN. The second is that all neurons, weights and connections have a number inside of them which shows their position in the array that they are allocated in, the arrays are implicit shown by colored boxes. The third is that only pointers to the first connection and weight connected to a neuron is required, because the neuron knows how many incoming connections it has.

4.3 Algorithmic Design

The algorithm for executing the network and the backpropagation algorithm is described in section 2.2.2 and 2.3.1 and the optimizations needed in these algorithms are described in section 3.3.

These algorithms are however not the only algorithms needed in this library.

4.3.1 The Density Algorithm

In section 3.1 I expressed the wish for a density parameter in the constructor. The reason for adding this parameter, is to allow it to be like a cheap version of the optimal brain damage algorithm [LeCun et al., 1990]. In large ANNs, many of the rules that are generated only utilize a few connections. Optimal brain damage try to remove unused connections. I will provide the opportunity to remove some connections in advance and then try to see if the network could be trained to give good results. There is no guarantee that this will give a better ANN, but it will give the users of the library another parameter to tweak in order to get better performance.

The algorithm, which creates an ANN with a certain density D, has a number of requirements which it must comply to:

1. The number of neurons should be the same as stated by the parameters to the constructor.
2. The bias neurons should still be fully connected, since they represent the t value in the activation functions. These connections should not be counted when calculating the density.
3. All neurons should be connected to at least one neuron in the previous layer and one neuron in the next layer.
4. The same connection must not occur twice.
5. The connections should be as random as possible.
6. The density in each layer should be as close to D as possible.

The algorithm that I have constructed to do this is illustrated in algorithm 1. Because some connections should always be in place, the actual connection rate may be different than the connection rate parameter given.

Algorithm 1 Algorithm for generating connections with a given density D in an ANN. Bias neurons is not represented in this algorithm in order to make it more simple.

Require: $0 \leq D \leq 1$
  {First find out how many input connections each neuron should have}
  for all $l_i,\ l_o$ where l_i is a layer with l_o as the next layer **do**
    $num_{in} \leftarrow$ the number of neurons in l_i
    $num_{out} \leftarrow$ the number of neurons in l_o
    $num_{min} \leftarrow max(num_{in},\ num_{out})$
    $num_{max} \leftarrow num_{in} * num_{out}$
    $num_{con} \leftarrow max(D * num_{max},\ num_{min})$
    Spread num_{con} equally among the neurons in l_o
  end for

  {Then make the actual connections}
  for all $l_i,\ l_o$ where l_i is a layer with l_o as the next layer **do**
    {Connections from all neurons in l_i}
    for all n_i where n_i is a neuron in l_i **do**
      $n_o \leftarrow$ a random neuron in l_o with room for more connections
      Make a connection between n_i and n_o
    end for

    {The rest of the connections}
    for all n_o where n_o is a neuron in l_o **do**
      while There is still room for connections in n_o **do**
        $n_i \leftarrow$ a random neuron in l_i which is not already connected to n_o
        Make a connection between n_i and n_o
      end while
    end for
  end for

4.3.2 The Activation Functions

I section 3.3.8 I suggested that a lookup table would probably be faster than calculating the actual activation functions. I have chosen not to implement this solution, but it would probably be a good idea to try it in the future.

This optimization has not been implemented because the activation function is only calculated once for each neuron and not once for each connection. The cost of the activation function becomes insignificant compared to the cost of the sum function, if fully connected ANNs become large enough.

The activation functions, that I have chosen to implement is the threshold function and the sigmoid function.

4.4 Fixed Point Design

In section 3.2 I explained, that after fully training an ANN you can make one check, that will guarantee, that there will be no integer overflow. In this section I will suggest several ways that this could be implemented and describe how I have implemented one of these suggestions.

The general idea behind all of these suggestions, is the fact that you can calculate the maximum value that you will ever get as input to the activation function. This is done, by assuming that the input on each connection into a neuron is the worst possible and then calculating how high a value you could get.

Since all inputs internally in the network is an output from another activation function, you will always know which value is the worst possible value. The inputs on the other hand are not controlled in any way. In order to ensure that an integer overflow will never occur, you will have to put constraints on the inputs. There are several different kinds of constraints you could put on the inputs, but I think that it would be beneficial to put the same constraints on the inputs as on the outputs, implying that the input should be between zero and one. Actually this constraint can be relaxed to allow inputs between minus one and one.

I now have full control of all the variables and I will have to choose an implementation method for ensuring that integer overflow will not occur. There are two questions here which needs to be answered. The first is how should the decimal point be handled? But I will come to this in section 4.4.1.

The second is when and how the check be should made? An obvious choice would be to let the fixed point library make the check, as this library needs the fixed point functionality. However, this presents a problem because the fixed point library is often run on some kind of portable or embedded device. The floating point library, on the other hand, is often run on a standard workstation. This fact suggests that it would be useful to let the floating point library do all the hard work and simply let the fixed point library read a configuration file saved by the floating point library.

These choices make for a model, where the floating point library trains the ANN, then checks for the possibility of integer overflow and saves the ANN in a fixed point version. The fixed point library then reads this configuration file and can begin executing inputs.

4.4.1 The Position of the Decimal Point

The position of the decimal point is the number of bits you will use for the fractional part of the fixed point number. The position of the decimal point also determines how many bits that can be used by the integer part of the fixed point number.

There are two ways of determining the position of the decimal point. The first way, is to set the decimal point at compile time. The second is to set the decimal point when saving the ANN from the floating point library.

There are several advantages of setting the decimal point at compile time:

- Easy to implement
- The precision is known in advance
- The scope of the inputs and outputs are known when writing the software using the fixed point library
- The compiler can optimize on basis of the decimal point

There are however also several advantages of setting the decimal point when saving the ANN to a fixed point configuration file:

- The precision will be as high as possible
- Less ANNs will fail the check that ensures that an overflow will not occur

Although there are less advantages in the last solution, it is the most general and scalable solution, therefore I will choose to set the decimal point when saving the ANN. The big question is now, where should the decimal point be? And how

can you be absolutely sure that an integer overflow will not occur when the decimal point is there?

Before calculating where the decimal point should be, I will define what happens with the number of used bits under certain conditions: When multiplying two integer numbers, you will need the same number of bits to represent the result, as was needed to represent both of the multipliers. E.g. if you multiply two 8 bit numbers, you will get a 16 bit number [Pendleton, 1993]. Furthermore, when adding or subtracting two signed integers you will need as many bits as used in the largest of the two numbers plus one. When doing a fixed point division, you will need to shift the numerator left by as many bits as the decimal point. When doing a decimal point multiplication, you first do a standard integer multiplication and then shift right by as many bits as the decimal point.

Several operations has the possibility of generating integer overflow, when looking at what happens when executing an ANN with equation 2.5.

If t is the number of bits needed for the fractional part of the fixed point number, we can calculate how many bits are needed for each of these operations. To help in these calculations, i will define the $y = bits(x)$ function, were y is the number of bits used to represent the integer part of x.

When calculating $w_i x_i$ we do a fixed point multiplication with a weight and a number between zero and one. The number of bits needed in this calculation is calculated in equation 4.1.

$$\begin{array}{lcl} t + bits(w_i) + t + bits(x_i) & = & \\ t + bits(w_i) + t + 0 & = & \\ 2t + bits(w_i) & & \end{array} \tag{4.1}$$

When calculating the activation function in fixed point numbers, it is calculated as a stepwise linear function. In this function the dominating calculation is a multiplication between a fixed point number between zero and one and the input to the activation function. From the input to the activation function another fixed point number is subtracted before the multiplication. The number of bits needed in this calculation is calculated in equation 4.2.

$$2t + bits\left(w_{n+1} \sum_{i=0}^{n} w_i x_i\right) + 1 \tag{4.2}$$

Since the highest possible output of the sum function is higher than the highest possible weight, this operation is the dominating operation. This implies, that if I can prove that an overflow will not occur in this operation, I can guarantee that an overflow will never occur.

When saving the ANN in the fixed point format, I can calculate the number of bits used for the integer part of the largest possible value that the activation function can be given as a parameter. This value is named m in equation 4.3, which calculates the position of the decimal point f for a n bit signed integer (remembering that one bit is needed for the sign).

$$f = \frac{n - 2 - m}{2} \tag{4.3}$$

4.5 Code Design

Section 3.3 describes the optimization techniques used when implementing the library. This section will not elaborate on this, but rather explain which techniques is used in the low-level code design, in order to make the library easy to use and maintain.

The library is written in ANSI C which puts some limitations on a programmer like me, who normally codes in C++. However, this is not an excuse for writing ugly code, although allocating all variables at the beginning of a function will never be pretty.

I have tried to use comprehensible variable and function names, I have also tried only using standard C functions, in order to make the library as portable as possible. I have made sure that I did not create any global variables and that all global functions or macros was named in a way, so that they would not easily interfere with other libraries (by adding the name fann). Furthermore, I have defined some macros which are defined differently in the fixed point version and in the floating point version. These macros help writing comprehensible code without too many `#ifdef`'s.

5 User's Guide

In this section I will describe how the intended use of this library is. For most usage the "Getting Started" section 5.2 should be sufficient, but for users with more advanced needs I will also recommend the "Advanced Usage" section 5.3.

The "Fixed Point Usage" section 5.4 is only intended for users with need of running the ANN on a computer with no floating point processor like e.g. an iPAQ.

5.1 Installation and Test

The library is developed on a Linux PC using the gcc compiler, but it should also be possible to compile the library on other platforms. Since the library is written as a part of this report, my main concern have not been to create an easy to use install method, but I plan to create one in the future.

In order to compile and test the library, go to the `src` directory and type `make runtest`. This will compile the library and run a couple of tests. An example output from this run is shown in appendix A.1. The output is quite verbose, but everything should work fine if the "`Test failed`" string is not shown in any of the last five lines.
If the test succeeds, the following libraries should be ready for use:

- `libfloatfann.a` The standard floating point library.
- `libdebugfloatfann.a` The standard floating point library, with debug output.
- `libdoublefann.a` The floating point library with double precision floats.
- `libdebugdoublefann.a` The floating point library with double precision floats and debug output.
- `libfixedfann.a` The fixed point library.
- `libdebugfixedfann.a` The fixed point library with debug output.

These libraries can either be used directly from this directory, or installed in other directories like e.g. `/usr/lib/`.

5.2 Getting Started

An ANN is normally run in two different modes, a training mode and an execution mode. Although it is possible to do this in the same program, I will recommend doing it in two different programs.

There are several reasons to why it is usually a good idea to write the training and execution in two different programs, but the most obvious is the fact that a typical ANN system is only trained once, while it is executed many times.

5.2.1 The Training

Figure 9 shows a simple program which trains an ANN with a data set and then saves the ANN to a file. The data is for the binary function XOR and is shown in figure 10, but it could have been data representing all kinds of problems.

Four functions are used in this program and often these are the only four functions you will need, when you train an ANN. I will now explain how each of these functions work.

fann_create Creates the ANN with a connection rate (1 for a fully connected network), a learning rate (0.7 is a reasonable default) and a parameter telling how many layers the network should consist of (including the input and output layer). After this parameter follows one parameter for each layer (starting with the input layer) telling how many neurons should be in each layer.

```
#include "floatfann.h"

int main()
{
        const float connection_rate = 1;
        const float learning_rate = 0.7;
        const unsigned int num_layers = 3;
        const unsigned int num_input = 2;
        const unsigned int num_neurons_hidden = 4;
        const unsigned int num_output = 1;
        const float desired_error = 0.0001;
        const unsigned int max_epochs = 500000;
        const unsigned int epochs_between_reports = 1000;

        struct fann *ann = fann_create(connection_rate,
                learning_rate, num_layers,
                num_input, num_neurons_hidden, num_output);

        fann_train_on_file(ann, "xor.data", max_epochs,
                epochs_between_reports, desired_error);

        fann_save(ann, "xor_float.net");

        fann_destroy(ann);

        return 0;
}
```

Figure 9: Simple program for training an ANN on the data in `xor.data` and saving the network in `xor_float.net`.

```
4 2 1
0 0
0
0 1
1
1 0
1
1 1
0
```

Figure 10: The file `xor.data`, used to train the xor function. The first line consists of three numbers: The first is the number of training pairs in the file, the second is the number of inputs and the third is the number of outputs. The rest of the file is the actual training data, consisting of one line with inputs, one with outputs etc.

fann_train_on_file Trains the ANN for a maximum of `max_epochs` epochs[5], or until the mean square error is lower than `desired_error`. A status line is written every `epochs_between_reports` epoch.

fann_save Saves the ANN to a file.

fann_destroy Destroys the ANN and deallocates the memory it uses.

The configuration file saved by `fann_save` contains all information needed in order to recreate the network. For more specific information about how it is stored please look in the source code.

5.2.2 The Execution

Figure 11 shows a simple program which executes a single input on the ANN, the output from this program can be seen in figure 12. The program introduces two new functions which was not used in the training procedure and it also introduces the `fann_type` type. I will now explain the two functions and the type:

fann_create_from_file Creates the network from a configuration file, which have earlier been saved by the training program in figure 9.

[5]During one epoch each of the training pairs are trained for one iteration.

```
#include <stdio.h>
#include "floatfann.h"

int main()
{
        fann_type *calc_out;
        fann_type input[2];

        struct fann *ann = fann_create_from_file("xor_float.net");

        input[0] = 0;
        input[1] = 1;
        calc_out = fann_run(ann, input);

        printf("xor test (%f,%f) -> %f\n",
                input[0], input[1], calc_out[0]);

        fann_destroy(ann);
        return 0;
}
```

Figure 11: Creates an ANN from the file `xor_float.net` and runs one array of inputs through the ANN.

```
xor test (0.000000,1.000000) -> 0.990685
```

Figure 12: The output from the program seen in figure 11.

fann_run Executes the input on the ANN and returns the output from the ANN.

fann_type Is the type used internally by the fann library. This type is `float` when including `floatfann.h`, `double` when including `doublefann.h` and `int` when including `fixedfann.h`. For further info on `fixedfann.h`, see section 5.4.

These six functions and one type described in these two sections, are all you will need to use the fann library. However, if you would like to exploit the full potential of the fann library, I suggest you read the "Advanced Usage" section and preferably the rest of this report.

5.3 Advanced Usage

In this section I will describe some of the low-level functions and how they can be used to obtain more control of the fann library. For a full list of functions, please see `fann.h` (appendix B.1.1), which has an explanation of all the fann library functions. Also feel free to take a look at the rest of the source code.

I will describe four different procedures, which can help to get more power out of the fann library: "Adjusting Parameters", "Network Design", "Understanding the Error-value" and "Training and Testing".

5.3.1 Adjusting Parameters

Several different parameters exists in an ANN, these parameters are given defaults in the fann library, but they can be adjusted at runtime. There is no sense in adjusting most of these parameters after the training, since it would invalidate the training, but it does make sense to adjust some of the parameters during training, as I will describe in section 5.3.4. Generally speaking, these are parameters that should be adjusted before training.

The learning rate, as described in equation 2.11, is one of the most important parameters, but unfortunately it is also a parameter which is hard to find a reasonable default for. I have several times ended up using 0.7, but it is a good idea to test several different learning rates when training a network. The

learning rate can be set when creating the network, but it can also be set by the `fann_set_learning_rate(struct fann *ann, float learning_rate)` function.

The initial weights are random values between -0.1 and 0.1, if other weights are preferred, the weights can be altered by the `void fann_randomize_weights(struct fann *ann, fann_type min_weight, fann_type max_weight)` function.

The standard activation function is the sigmoid activation function, but it is also possible to use the threshold activation function. I hope to add more activation functions in the future, but for now these will do. The two activation functions are defined as `FANN_SIGMOID` and `FANN_THRESHOLD` and are chosen by the two functions:

- `void fann_set_activation_function_hidden(struct fann *ann, unsigned int activation_function)`
- `void fann_set_activation_function_output(struct fann *ann, unsigned int activation_function)`

These two functions set the activation function for the hidden layers and for the output layer. Likewise the steepness parameter used in the sigmoid function can be adjusted by these two functions:

- `void fann_set_activation_hidden_steepness(struct fann *ann, fann_type steepness)`
- `void fann_set_activation_output_steepness(struct fann *ann, fann_type steepness)`

I have chosen to distinguish between the hidden layers and the output layer, to allow more flexibility. This is especially a good idea for users wanting discrete output from the network, since they can set the activation function for the output to threshold. Please note, that it is not possible to train a network when using the threshold activation function, due to the fact, that it is not differentiable. For more information about activation functions please see section 2.2.1.

5.3.2 Network Design

When creating a network it is necessary to define how many layers, neurons and connections it should have. If the network become too large, the ANN will have difficulties learning and when it does learn it will tend to over-fit resulting in poor generalization. If the network becomes too small, it will not be able to represent the rules needed to learn the problem and it will never gain a sufficiently low error rate.

The number of hidden layers is also important. Generally speaking, if the problem is simple it is often enough to have one or two hidden layers, but as the problems get more complex, so does the need for more layers.

One way of getting a large network which is not too complex, is to adjust the `connection_rate` parameter given to `fann_create`. If this parameter is 0.5, the constructed network will have the same amount of neurons, but only half as many connections. It is difficult to say which problems this approach is useful for, but if you have a problem which can be solved by a fully connected network, then it would be a good idea to see if it still works after removing half the connections.

5.3.3 Understanding the Error-value

The mean square error value is calculated while the ANN is being trained. Some functions are implemented, to use and manipulate this error value. The `float fann_get_error(struct fann *ann)` function returns the error value and the `void fann_reset_error( struct fann *ann)` resets the error value. I will now explain

how the mean square error value is calculated, to give an idea of the value's ability to reveal the quality of the training.

If d is the desired output of an output neuron and y is the actual output of the neuron, the square error is $(d - y)^2$. If two output neurons exists, then the mean square error for these two neurons is the average of the two square errors.

When training with the `fann_train_on_file` function, an error value is printed. This error value is the mean square error for all the training data. Meaning that it is the average of all the square errors in each of the training pairs.

5.3.4 Training and Testing

Normally it will be sufficient to use the `fann_train_on_file` training function, but some times you want to have more control and you will have to write a custom training loop. This could be because you would like another stop criteria, or because you would like to adjust some of the parameters during training. Another stop criteria than the value of the combined mean square error could be that each of the training pairs should have a mean square error lower than a given value.

```
struct fann_train_data *data = fann_read_train_from_file(filename);
for(i = 1; i <= max_epochs; i++){
        fann_reset_error(ann);
        for(j = 0; j != data->num_data; j++){
                fann_train(ann, data->input[j], data->output[j]);
        }
        if(fann_get_error(ann) < desired_error){
                break;
        }
}
fann_reset_error(ann);
fann_destroy_train(data);
```

Figure 13: The internals of the `fann_train_on_file` function, without writing the status line.

The internals of the `fann_train_on_file` function is shown in a simplified form in figure 13. This piece of code introduces the `void fann_train(struct fann *ann, fann_type *input, fann_type *desired_output)` function, which trains the ANN for one iteration with one pair of inputs and outputs and also updates the mean square error. The `fann_train_data` structure is also introduced, this structure is a container for the training data in the file described in figure 10. The structure can be used to train the ANN, but it can also be used to test the ANN with data which it has not been trained with.

```
struct fann_train_data *data = fann_read_train_from_file(filename);
fann_reset_error(ann);
for(i = 0; i != data->num_data; i++){
        fann_test(ann, data->input[i], data->output[i]);
}
printf("Mean Square Error: %f\n", fann_get_error(ann));
fann_destroy_train(data);
```

Figure 14: Test all of the data in a file and calculates the mean square error.

Figure 14 shows how the mean square error for a test file can be calculated. This piece of code introduces another useful function: `fann_type *fann_test(struct fann *ann, fann_type *input, fann_type *desired_output )`. This function takes an input array and a desired output array as the parameters and returns the calculated output. It also updates the mean square error.

5.3.5 Avoid Over-fitting

With the knowledge of how to train and test an ANN, a new approach to training can be introduced. If too much training is applied to a set of data, the ANN will eventually over-fit, meaning that it will be fitted precisely to this set of training data and thereby loosing generalization. It is often a good idea to test, how good an ANN performs on data that it has not seen before. Testing with data not seen before, can be done while training, to see how much training is required in order to perform well without over-fitting. The testing can either be done by hand, or an automatic test can be applied, which stops the training when the mean square error of the test data is not improving anymore.

5.3.6 Adjusting Parameters During Training

If a very low mean square error is required it can sometimes be a good idea to gradually decrease the learning rate during training, in order to make the adjusting of weights more subtle. If more precision is required, it might also be a good idea to use double precision floats instead of standard floats.

The threshold activation function is faster than the sigmoid function, but since it is not possible to train with this function, I will suggest another approach:

While training the ANN you could slightly increase the steepness parameter of the sigmoid function. This would make the sigmoid function more steep and make it look more like the threshold function. After this training session you could set the activation function to the threshold function and the ANN would work with this activation function. This approach will not work on all kinds of problems, but I have successfully tested it on the XOR function. The source code for this can be seen in appendix B.2.3

5.4 Fixed Point Usage

It is possible to run the ANN with fixed point numbers (internally represented as integers). This option is only intended for use on computers with no floating point processor, like e.g. the iPAQ, but a minor performance enhancement can also be seen on most modern computers (see section 6 "Benchmarks" for further info of the performance of this library). With this in mind, I will now describe how you should use the fixed point version of the fann library. If you do not know, how fixed point numbers work, please read section 3.2 and section 4.4.

5.4.1 Training a Fixed Point ANN

The ANN cannot be trained in fixed point, which is why the training part is basically the same as for floating point numbers. The only difference is that you should save the ANN as fixed point. This is done by the `int fann_save_to_fixed(struct fann *ann, const char *configuration_file)` function. This function saves a fixed point version of the ANN, but it also does some analysis, in order to find out where the decimal point should be. The result of this analysis is returned from the function.

The decimal point returned from the function is an indicator of, how many bits is used for the fractional part of the fixed point numbers. If this number is negative, there will most likely be integer overflow when running the library with fixed point numbers and this should be avoided. Furthermore, if the decimal point is too low (e.g. lower than 5), it is probably not a good idea to use the fixed point version.

Please note, that the inputs to networks that should be used in fixed point should be between -1 and 1.

An example of a program written to support training in both fixed point and floating point numbers is given in appendix B.2.1 `xor_train.c`.

5.4.2 Running a Fixed Point ANN

Running a fixed point ANN is done much like running an ordinary ANN. The difference is that the inputs and outputs should be in fixed point representation. Furthermore the inputs should be restricted to be between $-multiplier$ and $+multiplier$ to avoid integer overflow, where the *multiplier* is the value returned from `unsigned int fann_get_multiplier(struct fann *ann)`. This multiplier is the value that a floating point number should be multiplied with, in order to be a fixed point number, likewise the output of the ANN should be divided by this multiplier in order to be between zero and one.

To help using fixed point numbers, another function is provided. `unsigned int fann_get_decimal_point(struct fann *ann)` which returns the decimal point. The decimal point is the position dividing the integer and fractional part of the fixed point number and is useful for doing operations on the fixed point inputs and outputs.

For an example of a program written to support both fixed point and floating point numbers please see `xor_test.c` in appendix B.2.2.

5.4.3 Precision of a Fixed Point ANN

The fixed point ANN is not as precise as a floating point ANN, furthermore it approximates the sigmoid function by a stepwise linear function. Therefore, it is always a good idea to test the fixed point ANN after loading it from a file. This can be done by calculating the mean square error as described in figure 14. There is, however, one problem with this approach: The training data stored in the file is in floating point format. Therefore, it is possible to save this data in a fixed point format from within the floating point program. This is done by the function `void fann_save_train_to_fixed(struct fann_train_data* data, char *filename, unsigned int decimal_point)`. Please note that this function takes the decimal point as an argument, meaning that the decimal point should be calculated first by the `fann_save_to_fixed` function.

6 Benchmarks

In this section I will run several benchmarks on different ANN libraries in order to find out how well the libraries performs. In addition to this I will test the performance of different versions of the fann library. The benchmarks are divided in two parts:

Quality Benchmark This benchmark tests how good the libraries are at training different ANNs. The benchmark measures the mean square error over time.

Performance Benchmark This benchmark tests the performance of the libraries. The performance is tested on different sizes of ANNs in order to check how well the libraries scale. The performance is measured as execution-time per connection.

The reason for this division is the fact that it will not do any good to have a fast library, if it is not good at training.

The libraries will be tested both for quality and for performance. Part of this test will be on a hand-held Compaq iPAQ H3600 with a 206 MHz Intel StrongARM processor, which excels in the fact that it does not have a floating point processor and only has a 8 KB cache consisting of 256 cache lines of 32 bytes. The rest of the benchmarks will be run on a workstation AMD Athlon XP 1600+ machine (actually only 1400 MHz) with a 256 KB L2 cache and a 128 KB L1 cache. Both machines use the Linux operating system.

6.1 The Libraries

Besides the fann library, I will benchmark Jet's Neural Library [Heller, 2002] (hereafter known as jneural) and Lightweight Neural Network [van Rossum, 2003] (hereafter known as lwnn). I have made sure that all three libraries have been compiled with the same compiler and the same compile-options.

I have downloaded several other ANN libraries, but most of them had some problem making them difficult to use. Either they where not libraries, but programs [Anguita, 1993], [Zell, 2003], they could not compile [Software, 2002], or the documentation was so inadequate that is was not possible to implement the features needed in the benchmark [Darrington, 2003].

Even though I will only benchmark two libraries besides the fann library, I still think that they give a good coverage of the different libraries which are available. I will now briefly discuss the pros and cons of these two libraries.

6.1.1 Jet's Neural Library (jneural)

The jneural library [Heller, 2002] is a C++ library which is pretty straightforward to use. It supports several different network architectures and several different activation functions, but no possibility of changing the steepness. It uses bias neurons and supports the standard backpropagation algorithm. Besides this, only a few helper functions are implemented. E.g. it is possible to save the weights of an ANN to a file, but not possible to save the structure. The library is accompanied by a reference manual and a few easy to understand examples.

Part of the jneural library is an architecture designed to load training data from a file. This feature is a big help when training an ANN and should be a feature included in all ANN libraries.

The library internally implements an ANN as a lot of linked objects, which make for very poor cache performance. In its original form jneural used double precision

floats, but I have altered it to use single precision floats, in order to compare it to the other libraries. The library used in the benchmarks is version 1.05 from 2002.

I have used this library on several occasions, most recent in [Nissen et al., 2003], where the library was trained on a normal PC and executed on an iPAQ.

6.1.2 Lightweight Neural Network (lwnn)

The lwnn [van Rossum, 2003] library is a C library written with the purpose of being lightweight and easy to use. It only supports multilayer feedforward networks and only one activation function (sigmoid with no possibility of setting the steepness). A slightly modified backpropagation algorithm with a momentum parameter is implemented. The library supports a wide array of helper functions for using, saving and loading the ANN, but there is no support for loading training data from a file. A short and simple reference manual accompanies the library along with some large examples.

The library is in active development and has gone from version 0.3 to version 0.6 while I have implemented the fann library. Version 0.6 is used in the benchmarks.

The library has a good compact architecture and is highly optimized. E.g. the sigmoid activation function is implemented as a lookup table.

6.2 Quality Benchmark

In this section I will measure the quality of an ANN implementation, as how low mean square error values it can produce during a fixed period of training. It is always hard to test the quality of an ANN implementation. How well a library performs on a given problem is a combination of a number of factors, including the initial weights, the training algorithm, the activation function and the parameters for this function. Especially the initial weights are tricky because they are set at random. For this reason two training sessions with the same data can give different results.

Another problem is finding good datasets. ANN libraries perform different on different datasets, meaning that just because one library is better at one problem does not mean that it is better at another problem. For this reason quality benchmarks should be run on several different datasets. The datasets themselves should include both training and testing sets, to allow checks for over-fitting.

It is possible to make artificial datasets, but they very seldom reflect the kind of problems ANNs are faced with when they are used. For this reason many different databases with real ANN problems have been created, [Blake and Merz, 1998] being the largest. I have looked at several different sources of datasets, before I decided to chose the datasets delivered by Proben1 [Prechelt, 1994]. Proben1 delivers 12 different datasets, divided in two different categories, classification and approximation. These problems have been benchmarked with several different training methods and suggestions for network sizes have been given. I chose the Proben1 datasets because the problems are very differentiated and well documented.

6.2.1 Benchmark Setup

I will benchmark the three libraries with a selection of datasets from the Proben1 datasets. I will select the sets on the basis, that they should be representative of the problems in Proben1. Unfortunately some of the problems in Proben1 have serious over-fitting problems. I will only include a few of these because often the library which is best at fitting, is also best at over-fitting. This is most often not a problem with the library itself, but a problem which should be corrected by stopping the training before too much over-fitting occurs. For the network sizes I will use the

sized suggested in [Prechelt, 1994] and for the learning rate I will use 0.7 for all the libraries. The characteristics of the data sets are shown in figure 15, but for further information on the datasets please consult [Prechelt, 1994].

Dataset name	Type	Dataset size	Inputs	Neurons in hidden	Outputs
building	a	4208	14	16	3
card	c	690	51	32	2
gene	c	3175	120	4 + 2	3
mushroom	c	8124	125	32	2
soybean	c	683	82	16 + 8	19
thyroid	c	7200	21	16 + 8	3

Figure 15: The datasets, where the type is either a (approximation) or c (classification) and 16 + 8 in "Neurons in hidden" indicates that there are two hidden layers with 16 neurons in the first and 8 in the second.

The Proben1 datasets are separated with 50% for training, 25% for validation and 25% for testing. I will however not do validation while training and have for this reason decided to use both the validation and test sets for testing.

I will do the quality benchmarks by training ANNs with training sets for a fixed period of 200 seconds. During this period I will regularly stop the time in order to write information about the mean square error for the training data and the testing data to a file. Preferably once every second but since I will only stop training between two epochs, this can not always be accomplished. To calculate the mean square error I will use the same function on all the different libraries, to make sure that differences in this calculation does not affect the result.

I will use this data to create graphs which plot the mean square error as a function of the time. I will create one graph for each of the datasets. On these graphs I will plot the training and testing error for the jneural library, the lwnn library, the fann library with single precision floats and the fann library where connection rate is set to 0.75[6]. I will only plot data for one training session with each library, but I will run other smaller training sessions to ensure that the selected session is representative.

The mean square error of the training and testing data, executed on the fixed point version of the fann library will also be included on these graphs. This however needs a little bit of explanation: Every time I stop the time to print the mean square error for the fann library, I will also save the ANN to a fixed point configuration file. After the training has finished I will read each of these configuration files, with the fixed point library and calculate the mean square error for the training and testing data. This info will then be plotted on the graph.

I will sum up all of the benchmarks in figure 22, to give a quick overview of how well the libraries performed. The programs used for benchmarking the quality of the libraries is included in appendix B.3.1 and B.3.2.

6.2.2 The Benchmarks

Figure 16 to 21 show the benchmark graphs the different problems. Some of the graphs are shown on logarithmic scales to make it easier to see the differences between the libraries. Some of the plots do not start and end at the same time. This is because I have only allowed for testing of the networks between epochs and since some of the training sets where quite large (especially mushroom), the first epoch could take several seconds.

[6]The actual connection rate may be different, because some connections should always be in place.

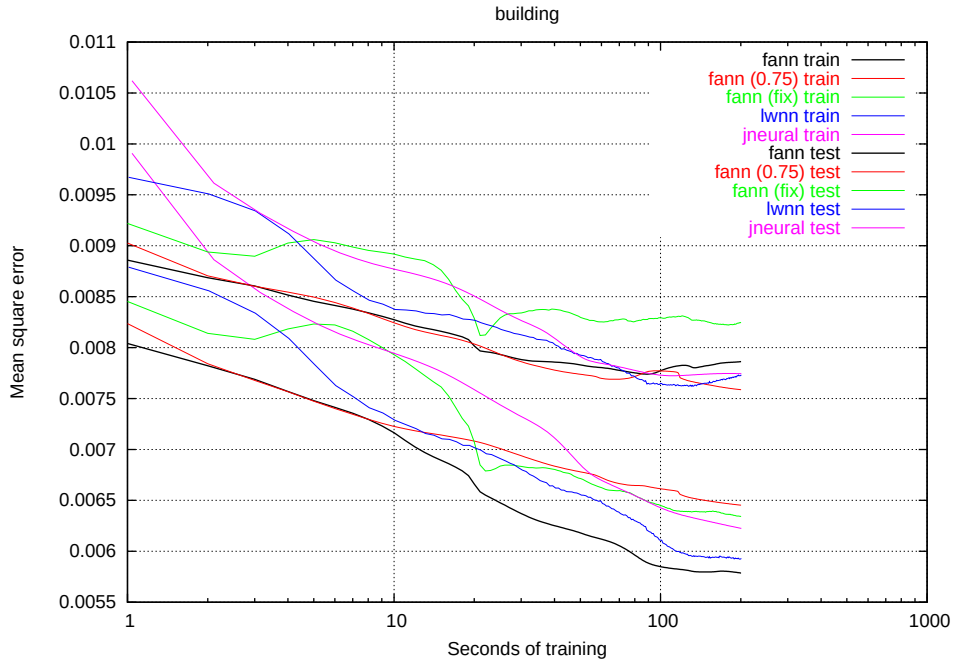


Figure 16: Graph showing the mean square error as a function of the time for the building problem. The mean square error drops in a stepwise manor, which is a typical behavior of training an ANN on difficult but solve-able problems.

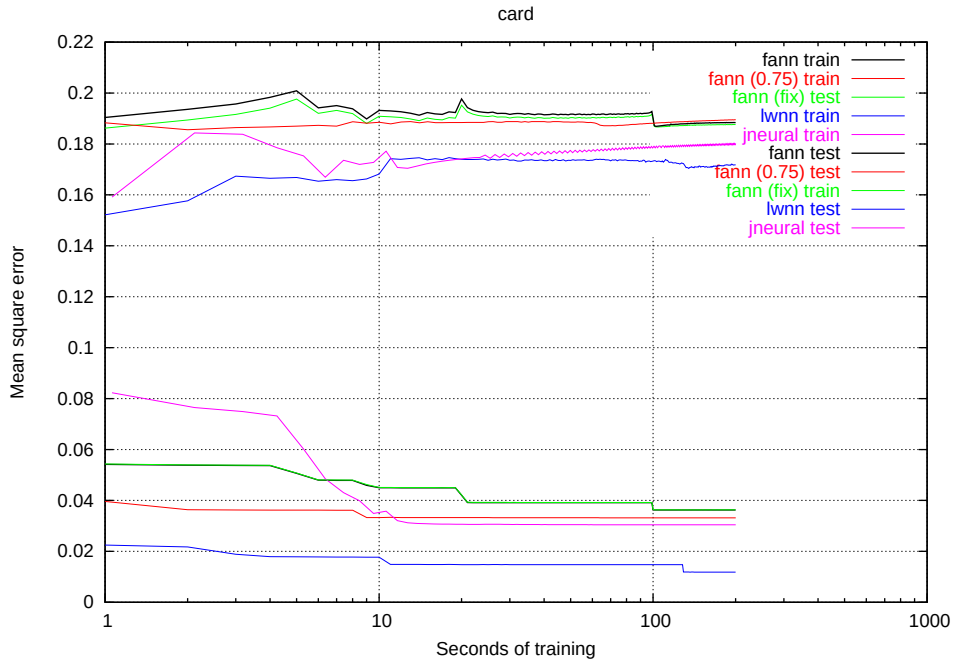


Figure 17: Graph showing the mean square error as a function of the time for the card problem. The libraries have difficulties training on this problem and hence the mean square error for the test data is high.

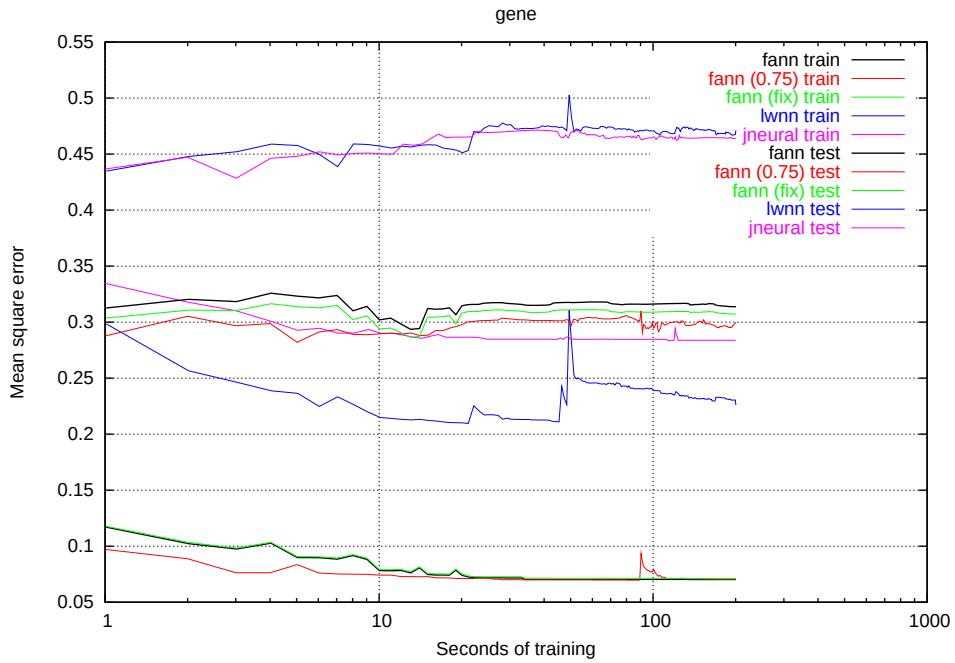


Figure 18: Graph showing the mean square error as a function of the time for the gene problem. It seems that the problem is difficult to train on and that there are some spikes in the optimization landscape.

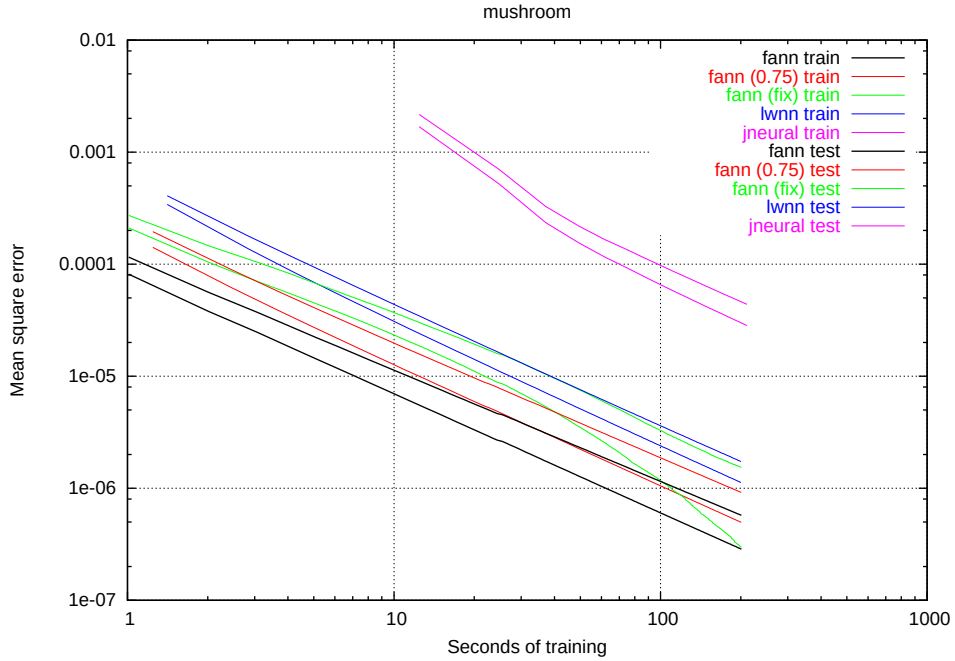


Figure 19: Graph showing the mean square error as a function of the time for the mushroom problem. Very easy problem to learn which generates straight lines on logarithmic scales.

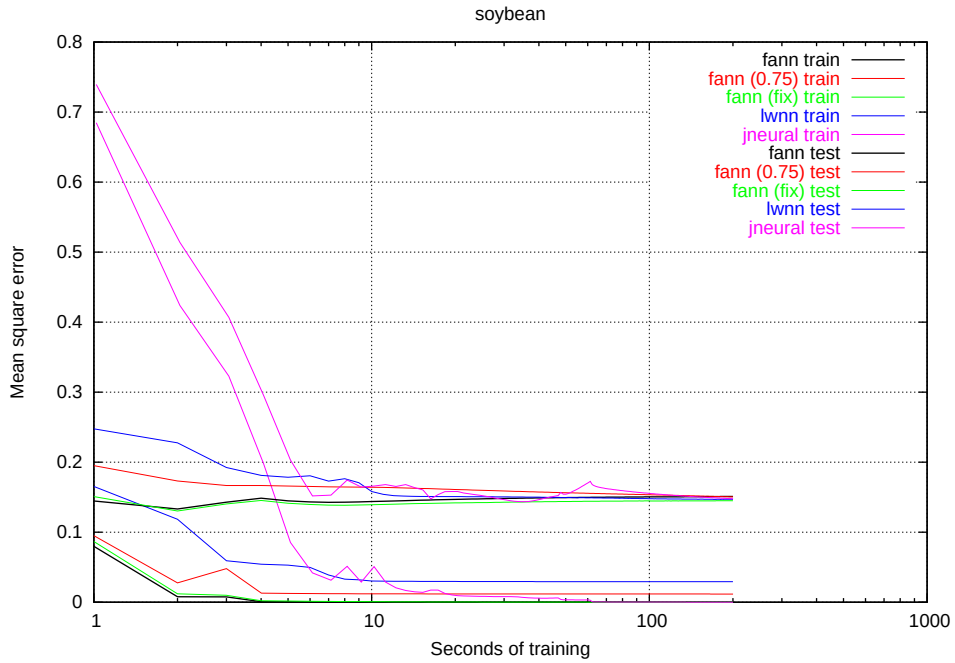


Figure 20: Graph showing the mean square error as a function of the time for the soybean problem. The training quickly converges to a standstill, with no more improvement.

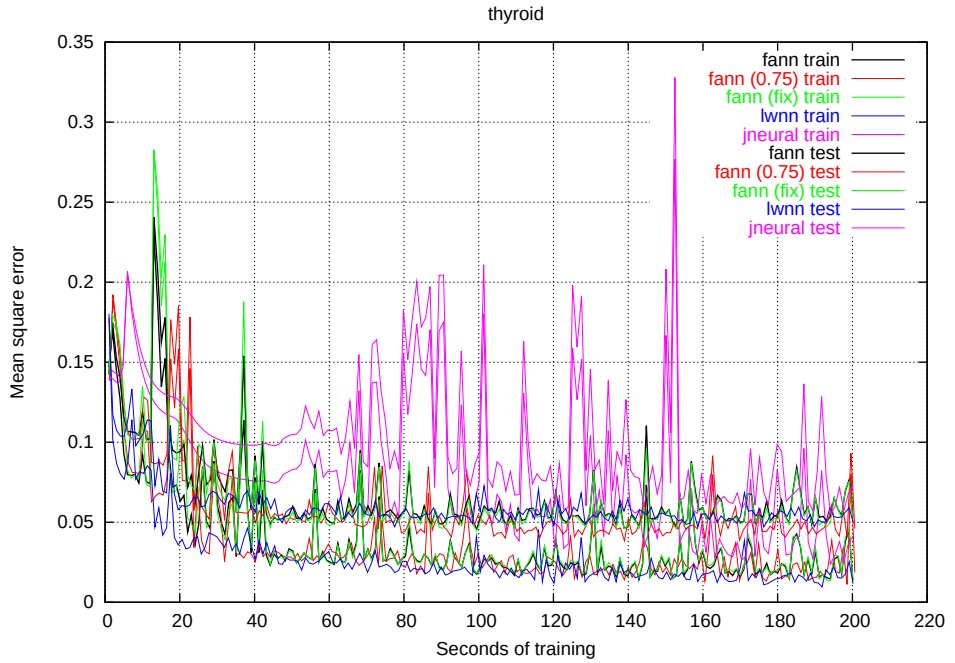


Figure 21: Graph showing the mean square error as a function of the time for the thyroid problem. A very spiked plot which is difficult to train on, but it seems that progress is made anyway.

It is expected that the test data performs worse than the training data, which is why the plot for the test data will be located higher than the plot for the training data. With this in mind, I have chosen to plot both test and training data in the same color to make the graphs easier to look at. I will now briefly discuss the results of the individual benchmarks.

Figure 16 - building This is a good problem, where training benefits both the mean square error of the training data and of the testing data. It seems that testing past the 200 seconds would be able to further decrease the mean square error.

Figure 17 - card This problem seems to over-fit easily. The problem has one input with the value 1.00112 in its training data. This violates the fixed point version of the library, since it only supports input between zero and one. It does, however, not seem to give rise to any problems.

It is worth noticing the fact that the fixed point version is slightly better than the floating point version. This behavior was repeated multiple times, but I can not find a reasonable explanation to why this behavior can be repeated. Some of the other benchmarks also show this behavior.

Figure 18 - gene This is the most difficult problem to solve, and it does not seem ideal for solving with ANNs. The jneural and the lwnn libraries seem to perform considerable worse on this problem than the fann library. I do not know why this problem is easier to solve for the fann library, but the behavior was repeatable. Perhaps the low number of neurons in the layers made it difficult to learn for the lwnn library, which does not use bias neurons. A clear spike is visible on the lwnn graph, which might suggest that the training is trying to get out of a local minima.

Figure 19 - mushroom This problem is very easy to solve for all the libraries. It seems like there exists a single rule which the libraries try to figure out and the more they train, the closer they come to the pure form of this rule. I would guess that this rule could be expressed in a simpler way than with an ANN.

Figure 20 - soybean The libraries quickly converge to a stand still and it seems that more training will not help this problem.

Figure 21 - thyroid This is a very tricky problem, which produces very spiked graphs. When comparing the spikes on the training data and the testing data for a library, there is a visible connection between how low the mean square errors are. This suggests that more training would in fact help, although it is difficult to determine when the training should stop.

The spiked look of the graphs suggests that perhaps gradient descent training methods are not the best way of training ANNs with this problem. Another optimization method like e.g. simulated annealing would probably do a better job.

6.2.3 Quality Benchmark Conclusion

Figure 22 shows a summary table of all the different runs and although there is not an obvious winner, there is an obvious looser. The jneural library is clearly slower than the other libraries and does not manage to reach as low a mean square error as the other libraries.

The lwnn library seems to be the second worst library, but much of this is due to the fact that it did a very poor job on the gene problem. When looking at the other problems it does a fairly good job (especially on the card problem). The library

Dataset	lwnn	jneural	fann	fann (0.75)	fann (fix)
building (train)	0.00593	0.00622	0.00579	0.00645	0.00634
building (test)	0.00773	0.00774	0.00786	0.00759	0.00828
card (train)	0.01185	0.03042	0.03624	0.03315	0.03628
card (test)	0.17171	0.18019	0.18839	0.18951	0.18761
gene (train)	0.22597	0.28381	0.07019	0.07057	0.07110
gene (test)	0.47122	0.46384	0.31386	0.29976	0.30720
mushroom (train)	1.13e-06	2.83e-05	2.86e-07	4.96e-07	2.96e-07
mushroom (test)	1.73e-06	4.39e-05	5.73e-07	9.18e-07	1.53e-06
soybean (train)	0.02927	6.16e-05	1.12e-05	0.01171	1.86e-05
soybean (test)	0.14692	0.14781	0.15114	0.15043	0.14500
thyroid (train)	0.01162	0.04124	0.01357	0.01857	0.01502
thyroid (test)	0.05013	0.07341	0.05347	0.04603	0.05313
Mean error train	0.04744	0.06030	0.02097	0.02341	0.02146
Mean error test	0.14129	0.14551	0.11912	0.11555	0.11687
Mean error total	0.09437	0.10290	0.07004	0.06948	0.06917
Total epochs	44995	9900	76591	67421	-

Figure 22: Summary of the quality benchmarks, showing the mean square error after the full training time. The library with the best mean square error is marked with green and the one with the worst is marked with red. Furthermore the total number of epochs for each library is shown.

is a little bit slower at learning than the fann library (44995 epochs compared to 76591), which explains why it does not get as low a mean square error for some of the easier problems like mushroom and building.

This leaves the three versions of the fann library: The standard fann library has the lowest mean square error for the training data, the fann library with a connection rate of 0.75 has the lowest mean square error for the testing data and the fixed point fann library has the lowest total mean square error.

The standard fann library does a really good job on most of the problems. It is also the library which manages to train for the most epochs. It only has one set of data, where it finishes last and this is the test data for the soybean problem. But the difference between the libraries on this set of data is so small that it really does not matter.

The fann library with a connection rate of 0.75, is a bit better at generalizing than the standard fann library, but there is no clear tendency. Surprisingly it does not manage to train for as many epochs as the standard fann library, but it is probably due to the fact that the standard fann library uses optimizations for fully connected networks.

To my big surprise, the overall winner was the fixed point version of the fann library. I will however say that it must be a coincidence that it is better than the standard fann library because the fixed point library uses the weights stored from the floating point library. It is however not a coincidence that the fixed point fann library is just as good as the floating point fann library. When looking at the datafiles saved by the floating point library, it is possible to see which positions of the decimal point that are used. Throughout all of the benchmarks the decimal point has been in the bit position range 10 - 13, these bit positions give plenty of accuracy to execute an ANN.

From these observations I can conclude that the fixed point implementation has proven that it can perform just as good as a floating point library. Though it is not possible to conclude that it will always perform this well. Some problems may give rise to very high weights, which will give a lower accuracy for the fixed point

library.

The final conclusion must be that both the lwnn library and the fann library does a good job on these problems. I suspect that with tweaking of parameters these two libraries will perform well on most ANN problems.

6.3 Performance Benchmark

In this section I will run two benchmarks with the lwnn library, the jneural library and several configurations of the fann library. Both benchmarks will measure the nanoseconds used per connection when executing different sizes of ANNs on the library.

The first benchmark will be run on the AMD Athlon machine and the second benchmark will be run on the iPAQ.

The configurations of the fann library which I will use are the normal fann library with the sigmoid activation function, the fann library with the threshold activation function (hereafter known as fann (thres)), the fixed point fann library (hereafter known as fann (fix)) and a version of the fann library which does not use the performance enhancements for fully connected networks (hereafter known as fann (noopt)). The reasons for choosing these configurations are:

fann (thres) This library measures how much the sigmoid activation function slows down the ANN.

fann (fix) Measures the performance enhancements produced by the fixed point optimizations.

fann (noopt) Measures the performance enhancements produced by the optimizations for fully connected networks. This can be used to see how small the connection rate should be before the ANN would be faster than a fully connected ANN.

I will measure the performance for several different sizes of ANNs. These different sized ANNs will consist of four layers with the same amount of neurons in each. The amount of neurons in the layers will be doubled for each run starting with only one neuron. With these four layers, the total amount of connections in an ANN without bias neurons (only lwnn) will be $3n^2$, where n is the amount of neurons in the layers. For an ANN with bias neurons (jneural and fann) the total amount of connections will be $3(n^2 + n)$.

For each network size, the network will be executed consecutive for 20 seconds. After this time the number of nanoseconds used for each connection will be calculated. The two benchmarks will produce one graph each, furthermore I will include tables showing the performance of layer sizes which are of particular interest. The program used for benchmarking the performance in the libraries is included in appendix B.3.3.

6.3.1 The Benchmark on the AMD Athlon

Figure 23 shows the benchmark for the libraries on the AMD Athlon machine. Before describing which library is the fastest, I will describe the shapes of the plots. All of the plots have a characteristic S shape, which start high then goes low before going high again. The reason why the plots start high is that, for small network sizes, the per neuron overhead is rather high compared to the time used on the connections. As ANNs become larger, the inner loops will run for more iterations, thus making the overhead smaller. If good cache optimization is applied, the ANN will run very smooth and all the CPU time will be used for actually calculating the

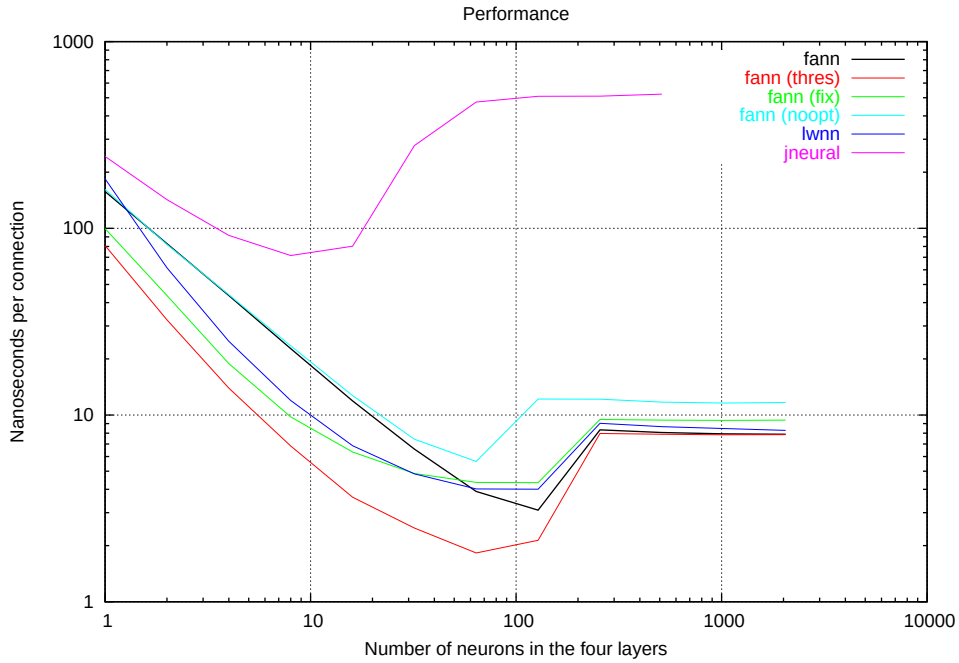


Figure 23: Graph showing the number of nanoseconds used per connection as a function of the network size, on an AMD Athlon machine.

sum function. At some point the ANNs become so large, that they can no longer fit in the cache, which on the AMD Athlon is 256 KB. When this happens, the performance declines and the plots go up again. The fann and lwnn library both hit the cache boundary between layer sizes of 128 and 256. This is the optimal place, since the amount of space needed to represent the weights in an ANN with layer sizes of 128 and 256 is approximately 190 KB and 770 KB (when using four bytes for each weight). The fann (noopt) configuration of the fann library hits the cache boundary before, because it needs to access memory containing information about the connections. Where the jneural library has no optimizations for cache and therefore hits the cache boundary a lot sooner.

A brief look at the graphs shows that the jneural library has the lowest performance and that the fann (thres) configuration performs best. The fann (noopt) configuration performs slightly worse than the three remaining libraries (lwnn, fann and fann (fix)), but between these libraries performance varies for different network sizes.

I have chosen three different layer sizes, which I will study in detail to see how fast the individual libraries are. For each of these sizes I have made a table showing how fast the libraries are.

Figure 24 shows a table for the performances of ANNs with 8 neurons in each layer. This is the layer size where the jneural library performs best. Unfortunately it still uses more than three times as much time as the rest of the libraries. The fann (noopt) configurations is only slightly slower than the fann configuration, which comes from the fact, that there is still plenty of free cache. The lwnn library is faster than the fann library, which is probably due to the fact that it uses a precalculated table for calculating the sigmoid function. Using a table for the sigmoid function greatly reduces the per neuron overhead and gives the lwnn library a performance increase on the small ANNs. The fixed point library performs quite well for these smaller problems and proves that the fixed point library can also increase

Library	Nanoseconds per neuron	Times better than jneural
fann	22.675	3.151
fann (thres)	6.827	10.465
fann (fix)	9.766	7.316
fann (noopt)	23.487	3.048
lwnn	11.962	5.972
jneural	71.440	1.000

Figure 24: The results with fully connected networks consisting of four layers with 8 neurons in each, on an AMD Athlon machine (this is the network size which gives the highest performance for the jneural library).

the performance on computers with floating point processors. This is probably because the stepwise linear activation function is faster than the sigmoid activation function. The fann (thres) configuration is the fastest configuration and it is more than three times as fast as the normal fann library, which suggests that the sigmoid function uses two third of the fann library's time. This observation clearly suggests, that creating a fast sigmoid implementation is a good way of optimizing an ANN library.

Library	Nanoseconds per neuron	Times faster than jneural
fann	3.0972	164.388
fann (thres)	2.1340	238.583
fann (fix)	4.344	117.194
fann (noopt)	12.175	41.818
lwnn	4.007	127.074
jneural	509.138	1.000

Figure 25: The results with fully connected networks consisting of four layers with 128 neurons in each, on an AMD Athlon machine (this is the network size which gives the highest performance for the lwnn and fann libraries).

Figure 25 shows how well the libraries performs with 128 neurons in each of the four layers. This is an indication of how the libraries perform at a stage where the jneural and fann (noopt) libraries have already reached the cache boundary, but the others have not. At this point the fann (thres) library is 238.583 times faster than the jneural library and the fann library is 164.388 times faster than the jneural library. This is a clear indication of the benefit which can be reached through performance engineering.

Another interesting observation is that the standard fann library is faster than the fann (fix) and lwnn libraries. I think this is due to the fact that both of these libraries use variables which are stored in memory for calculating their sigmoid functions. These variables will most likely have been erased from the cache at the point where the sigmoid function should be calculated, hence resulting in a cache miss. On the AMD Athlon a cache miss takes longer time than calculating the sigmoid function, which in effect makes the fann library faster than the two other libraries.

With a layer size of 512 neurons (figure 26), all of the libraries have reached the cache boundary. The jneural library is also close to the limit for how many connections it can handle without being killed by the system. I did not really investigate further into why it was killed by the system, but it probably took up too many resources. At this point the fann library is 64.867 times faster than the jneural library, which is not as much as before it reached the cache boundary, but it is still a huge performance increase.

Library	Nanoseconds per neuron	Times faster than jneural
fann	8.065	64.867
fann (thres)	7.890	66.310
fann (fix)	9.390	55.715
fann (noopt)	11.735	44.584
lwnn	8.662	60.400
jneural	523.175	1.000

Figure 26: The results with fully connected networks consisting of four layers with 512 neurons in each, on an AMD Athlon machine (this is the largest network size which could be executed by the jneural library).

With this large layer size, there is very little penalty of calculating the sigmoid function and the fann (thres) library is only slightly faster than the normal fann library. The lwnn library, which uses a table lookup for the sigmoid function, is now slower than the fann library and so is the fann (fix) library. The fann (noopt) library is still slower, but the difference is no longer as severe.

6.3.2 The Benchmark on the iPAQ

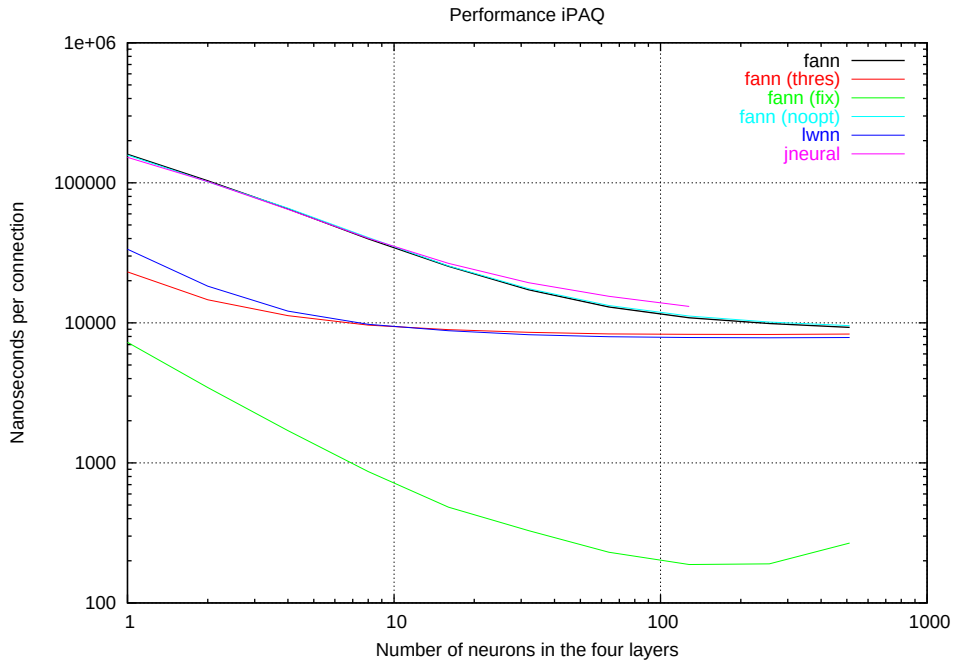


Figure 27: Graph showing the number of nanoseconds used per connection as a function of the network size, on an iPAQ.

Figure 27 shows the performance graph of the benchmarks run on the iPAQ. The first thing which is noticed, is the fact that the plots do not have an S shape. This is because there is only a small amount of cache on the iPAQ and that this cache is not suited for storing large arrays. Much of the cache is probably occupied by the operating system. The bottleneck on the iPAQ is the CPU running at 206 MHz and without a floating point unit. This makes the sigmoid function very hard to calculate and if we look at results for the smaller networks on the graph the plots are divided in three groups.

Library	Nanoseconds per neuron	Times faster than jneural
fann	39843.236	1.008
fann (thres)	9626.957	4.171
fann (fix)	867.798	46.272
fann (noopt)	40716.392	0.986
lwnn	9768.376	4.112
jneural	40154.518	1.000

Figure 28: The results with fully connected networks consisting of four layers with 8 neurons in each, on an iPAQ.

Figure 28 clearly shows the three groups that the plots are divided into. The first group consists of the fann, fann (noopt) and the jneural library. These three libraries all need to calculate the real sigmoid function. The second group consists of the fann (thres) and lwnn libraries which does not need to calculate the real sigmoid function and therefore is faster. The last group consists of the fann (fix) library, which is much faster than the other libraries because it uses fixed point numbers.

Library	Nanoseconds per neuron	Times faster than jneural
fann	10,858.245	1.207
fann (thres)	8,272.628	1.585
fann (fix)	188.134	69.677
fann (noopt)	11,178.446	1.173
lwnn	7,851.955	1.669
jneural	13,108.617	1.000

Figure 29: The results with fully connected networks consisting of four layers with 128 neurons in each, on an iPAQ (this is the largest network size which could be executed by the jneural library).

At a layer size of 128 (figure 29) the per neuron overhead is not as important and neither is the time it takes to calculate the sigmoid function. What is important, is how much the library uses floating point calculations. The fann (fix) library uses no floating point calculations when executing the library and hence receives a huge performance benefit. The fann (fix) library is almost 70 times faster than the jneural library and more than 40 times faster than the lwnn library, which really is a noticeable performance increase.

A surprising thing, which is visible on these figures, is how close the plot for the standard fann library is to the plot for the jneural library. This means that all the optimizations made to the fann library has very little influence on how fast it performs on the iPAQ. Had I not made the fixed point optimizations, then this would truly have been a depressing benchmark.

6.4 Benchmark Conclusion

The jneural library was clearly worse than the other libraries on almost all points of the benchmark tests. The fixed point implementation of the fann library proved its use by being both accurate and very fast on the iPAQ. A point worth noticing is seen when comparing figure 25 with figure 29. These two figures show how the libraries perform on fully connected ANNs with four layers and 128 neurons in each, but on different machines. On the iPAQ the fann (fix) library uses 188.134 nanoseconds per connection, while the jneural library uses 509.138 nanoseconds per connection on the AMD Athlon. This shows that the fixed point fann library is 2.7 times faster

on a 206 MHz hand-held iPAQ, than the jneural library is on a 1400 MHz AMD Athlon workstation.

It is difficult to tell which library is the fastest and most accurate, when comparing the standard fann library and the lwnn library. The lwnn library is faster at the smaller problems due to a highly optimized sigmoid function, while the standard fann library is faster at the larger problems due to optimized inner loops and cache performance. These benchmarks makes it easier to choose a library, when you know which kind of problem you should solve.

Many different optimizations where made in the fann and lwnn libraries, but when observing differences between figure 23 and figure 27, three optimizations appear to be most efficient.

Cache optimization This optimization is only effective on the AMD Athlon machine and gives the fann and lwnn libraries a huge advantage on this machine.

Fixed point optimization This optimization is only effective on the iPAQ and gives the fixed point fann library a huge advantage on this machine.

Fast sigmoid This optimization is effective on both machines and gives the lwnn and fixed point fann library a minor advantage.

A lot can be learned from these benchmarks. I will now discuss some of these learnings and suggest ways of improving the fann library on the basis of these learnings:

The sigmoid function takes a long time to calculate and optimizations on the time used for this function should be implemented in the fann library.

Sparse connected ANNs have serious performance problems in the fann library. For this reason they should be used with care, but still it seems like smaller sized ANNs could receive extra performance by decreasing the number of connections.

Cache performance is very important for the execution time of ANN libraries, perhaps further improvements could be received by looking even closer at the memory accessed by the fann library.

7 Conclusion

When I decided to develop the fann library, I had two reasons for developing it, the first reason was that I wanted to have a fast ANN library which could be used on our robot [Nissen et al., 2003]. The second reason was that I wanted to have full control of a fast ANN library, which could be used in my master thesis. As the project grew, a third reason became more and more evident, I wanted to make a library which would be used by other developers.

These three reasons for developing the fann library had different requirements. If the library should be used on the robot, it should be fast on systems with no floating point processor. If the library should be used in my master thesis, it should be fast on standard workstation machines and it should be very versatile to allow for new functionality to be implemented. If the library should be used by others, it should be both fast, versatile and easy to use.

I knew that it would not be easy to develop a fixed point version of the library, but I also knew that it could be done. During the development of the library several different methods of avoiding integer overflow was suggested (see section 4.4), but I think the method I finally chose was the most elegant. The fixed point fann library was a huge success in all the benchmarks, it was accurate in the quality benchmarks and it was more than 40 times faster than all the other libraries in the performance benchmark on the iPAQ.

The aim for the floating point library, was to be very fast while still allowing both fully and sparse connected ANNs to function. This proved a bigger challenge than first expected and my first implementation did not meet the demands (see section 4.2). The second implementation did however, meet the demands although it is only faster than the lwnn library in some situations. This shows, that although the library is fast, there is still room for improvement.

If the fann library should be used by other developers, speed is not the only requirement. Just as important requirements are, that the library should be easy to use and install, well documented and versatile. The fann library is clearly easy to use, as shown in figure 9 and figure 11. It is also versatile as shown in section 5.3, but what about, easy to install and well documented? No polished install system accompanies the library, hence installation is not easy, still it should not be too difficult to install the library. The user's guide documents how the library should be used, but it does not include a complete reference manual. Appendix B.1.1 `fann.h` can function as a reference manual, but it is not an ideal solution. In conclusion, the library could easily be used by other developers, but some work still needs to be done in order to make it appealing to a broad audience.

It is sometimes argued that performance engineering is not a real computer science, because new algorithms are not created. I do however not share this opinion and will now give a short example to illustrate the importance of performance engineering. On the AMD Athlon the jneural library uses 509.138 nanoseconds per connection in a fully connected ANN with four layers and 128 neurons in each, on the iPAQ the fixed point fann library uses 188.134 nanoseconds per connection on a similar ANN. This is 2.7 times faster than the jneural library and although it is probably possible to buy a computer, which will execute the jneural library just as fast as the fixed point library on the iPAQ, then it would not be possible to fit this computer into your pocket. This clearly illustrates that sometimes performance engineering is the only choice.

I am most satisfied with the result of this project, the requirements for the library where met and the performance of the library exceeded my expectations. It has been a fun challenge to develop the fann library and I look forward to releasing it on SourceForge.net.

7.1 Future Work

After the library has been released on SourceForge.net, a few changes will be made to make installation of the library easier, but numerous other additions could be made to the library. In this section I will discuss some of these additions.

Many different algorithms can be used for training ANNs, it would be nice to be able to choose from some of these algorithms in the fann library. As a first addition it would be nice have a better gradient descent algorithm like e.g. quickprop [Fahlman, 1988] or RPROP [Riedmiller and Braun, 1993], but later it would be nice to apply more general purpose optimization techniques like e.g. simulated annealing (perhaps this algorithm would do a better job on the thyroid problem).

At this point only two activation functions exists in the fann library (sigmoid and threshold). More activation functions could easily be added, like e.g. the hyperbolic tangent. This activation function has an output in the range of -1 to 1, while the other two activation functions has an output in the range of 0 to 1. Tests have shown that coding binary input as -1 and 1 is better than coding it as 0 and 1 [Sarle, 2002], which in effect would also make an ANN work better if the activation function used for the hidden layers gave an output in the range of -1 to 1. Since the sigmoid function can be altered to give an output in the range of -1 to 1, by the following equation: $y(x) = 2 \times sigmoid(x) - 1$, the range of the output from the activation functions could be a separate parameter, which is not determined by the type of activation function.

The cost of calculating the activation functions could be optimized. This could either be done by using the stepwise linear activation function as used in the fixed point library, or by using a table lookup as in the lwnn library.

It seems like fully connected ANNs are usually faster than sparse connected ANNs, it would however still be a good idea to add an optimal brain damage algorithm [LeCun et al., 1990] which could remove unused connections.

If the fann library should be used in developing a learning game bot for Quake III Arena, a reinforcement learning framework should be developed. This framework could either be a part of the fann library, or it could be a separate library which depend on the fann library for ANN support. I think developing the reinforcement learning framework as an independent library would be a good idea, to avoid confusion of what the primary objectives of the fann library are.

All of these additions add new functionality to the library, but other additions, which do not add functionality but rather improve the existing functionality, could also be proposed. Some of these additions could be:

- Add a C++ wrapper to allow a more object oriented approach, to using the library.
- Rewrite the inner loop the execution function in assembler (perhaps with use of the MMX instruction set), for extra performance.
- Add the possibility of more precision to the fixed point library, by allowing the users to chose the `long int` type as internal representation.
- The layer sizes are given as parameters to the `fann_create` function, in effect making the number of layers a parameter which should be known at compile time. It would be nice to add another create function which do not have these limitations.

Hopefully many of these additions will be implemented, I will try to implement some of them before I start my master thesis. During my master thesis I will probably implement some of the remaining additions and I will most likely also implement additions not mentioned here.

References

[Anderson, 1995] Anderson, J. A. (1995). *An Introduction to Neural Networks*. The MIT Press.

[Anguita, 1993] Anguita, D. (1993). Matrix back propagation v1.1.

[Bentley, 1982] Bentley, J. L. (1982). *Writing Efficient Programs*. Prentice-Hall.

[Blake and Merz, 1998] Blake, C. and Merz, C. (1998). UCI repository of machine learning databases.
http://www.ics.uci.edu/∼mlearn/MLRepository.html.

[Darrington, 2003] Darrington, J. (2003). Libann.
http://www.nongnu.org/libann/index.html.

[Fahlman, 1988] Fahlman, S. E. (1988). Faster-learning variations on back-propagation: An emperical study.

[FSF, 1999] FSF, F. S. F. (1999). Gnu lesser general public license.
http://www.fsf.org/copyleft/lesser.html.

[Hassoun, 1995] Hassoun, M. H. (1995). *Fundamentals of Artificial Neural Networks*. The MIT Press.

[Heller, 2002] Heller, J. (2002). Jet's neural library.
http://www.voltar.org/jneural/jneural_doc/.

[Hertz et al., 1991] Hertz, J., Krogh, A., and Palmer, R. G. (1991). *Introduction to The Theory of Neural Computing*. Addison-Wesley Publishing Company.

[IDS, 2000] IDS, I. S. (2000). Quake III arena.
http://www.idsoftware.com/games/quake/quake3-arena/.

[Kaelbling et al., 1996] Kaelbling, L. P., Littman, M. L., and Moore, A. P. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285.

[LeCun et al., 1990] LeCun, Y., Denker, J., Solla, S., Howard, R. E., and Jackel, L. D. (1990). Optimal brain damage. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems II*, San Mateo, CA. Morgan Kauffman.

[Nissen et al., 2003] Nissen, S., Damkjær, J., Hansson, J., Larsen, S., and Jensen, S. (2003). Real-time image processing of an ipaq based robot with fuzzy logic (fuzzy).
http://www.hamster.dk/˜purple/robot/fuzzy/weblog/.

[Nissen et al., 2002] Nissen, S., Larsen, S., and Jensen, S. (2002). Real-time image processing of an iPAQ based robot (iBOT).
http://www.hamster.dk/˜purple/robot/iBOT/report.pdf.

[OSDN, 2003] OSDN, O. S. D. N. (2003). Sourceforge.net.
http://sourceforge.net/.

[Pendleton, 1993] Pendleton, R. C. (1993). Doing it fast.
http://www.gameprogrammer.com/4-fixed.html.

[Prechelt, 1994] Prechelt, L. (1994). Proben1 – a set of neural network benchmark problems and benchmarking rules.

[Riedmiller and Braun, 1993] Riedmiller, M. and Braun, H. (1993). A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proc. of the IEEE Intl. Conf. on Neural Networks*, pages 586–591, San Francisco, CA.

[Sarle, 2002] Sarle, W. S. (2002). Neural network faq. ftp://ftp.sas.com/pub/neural/FAQ2.html#A_binary.

[Software, 2002] Software, W. (2002). Ann++. http://savannah.nongnu.org/projects/annpp/.

[Tettamanzi and Tomassini, 2001] Tettamanzi, A. and Tomassini, M. (2001). *Soft Computing*. Springer-Verlag.

[van Rossum, 2003] van Rossum, P. (2003). Lightweight neural network. http://lwneuralnet.sourceforge.net/.

[van Waveren, 2001] van Waveren, J. P. (2001). The quake III arena bot. http://www.kbs.twi.tudelft.nl/Publications/MSc/2001-VanWaveren-MSc.html.

[Zell, 2003] Zell, A. (2003). Stuttgart neural network simulator. http://www-ra.informatik.uni-tuebingen.de/SNNS/.

A Output from runs

A.1 Output from `make runtest`

```
Training network
./test/xor_train_float
Creating network.
creating network with learning rate 0.700000 and connection rate 1.000000
input
  layer       : 3 neurons, 1 bias
  layer       : 5 neurons, 1 bias
  layer       : 2 neurons, 1 bias
output
Training network.
Max epochs   500000. Desired error: 0.0001000000
Epochs            1. Current error: 0.2835713029
Epochs         1000. Current error: 0.0326964930
Epochs         2000. Current error: 0.0014186953
Epochs         3000. Current error: 0.0006593352
Epochs         4000. Current error: 0.0004221874
Epochs         5000. Current error: 0.0003083595
Epochs         6000. Current error: 0.0002419698
Epochs         7000. Current error: 0.0001986573
Epochs         8000. Current error: 0.0001682522
Epochs         9000. Current error: 0.0001457587
Epochs        10000. Current error: 0.0001284765
Epochs        11000. Current error: 0.0001147857
Epochs        12000. Current error: 0.0001036724
Epochs        12379. Current error: 0.0000999907
Testing network.
XOR test (0.000000,0.000000) -> 0.007620, should be 0.000000, difference=0.007620
XOR test (0.000000,1.000000) -> 0.990256, should be 1.000000, difference=0.009744
XOR test (1.000000,0.000000) -> 0.990946, should be 1.000000, difference=0.009054
XOR test (1.000000,1.000000) -> 0.012838, should be 0.000000, difference=0.012838
Saving network.
calculated_fix_point=12, fix_point=12, bits_used_for_max=5
Cleaning up.
Testing network with floats
./test/xor_test_float
Creating network.
creating network with learning rate 0.700000
input
  layer       : 3 neurons, 1 bias
  layer       : 5 neurons, 1 bias
  layer       : 2 neurons, 1 bias
output
Testing network.
XOR test (0.000000, 0.000000) -> 0.007620, should be 0.000000, difference=0.007620
XOR test (0.000000, 1.000000) -> 0.990256, should be 1.000000, difference=0.009744
XOR test (1.000000, 0.000000) -> 0.990946, should be 1.000000, difference=0.009054
XOR test (1.000000, 1.000000) -> 0.012838, should be 0.000000, difference=0.012838
Cleaning up.
Testing network with fixed points
./test/xor_test_fixed
Creating network.
creating network with learning rate 0.700000
input
  layer       : 3 neurons, 1 bias
  layer       : 5 neurons, 1 bias
  layer       : 2 neurons, 1 bias
output
Testing network.
XOR test (0, 0) -> 85, should be 0, difference=0.020752
XOR test (0, 4096) -> 3985, should be 4096, difference=0.027100
XOR test (4096, 0) -> 3991, should be 4096, difference=0.025635
XOR test (4096, 4096) -> 122, should be 0, difference=0.029785
Cleaning up.
```

B Source Code

B.1 The library

B.1.1 fann.h

```
/* This file defines the user interface to the fann library.
   It is included from fixedfann.h, floatfann.h and doublefann.h and should
   NOT be included directly.
*/

#include "fann_data.h"
#include "fann_internal.h"

#ifdef __cplusplus
extern "C" {
#endif


/* --- Initialisation and configuration --- */

/* Constructs a backpropagation neural network, from an connection rate,
   a learning rate, the number of layers and the number of neurons in each
   of the layers.

   The connection rate controls how many connections there will be in the
   network. If the connection rate is set to 1, the network will be fully
   connected, but if it is set to 0.5 only half of the connections will be set.

   There will be a bias neuron in each layer (except the output layer),
   and this bias neuron will be connected to all neurons in the next layer.
   When running the network, the bias nodes always emits 1
 */
struct fann * fann_create(float connection_rate, float learning_rate,
    /* the number of layers, including the input and output layer */
    unsigned int num_layers,
    /* the number of neurons in each of the layers, starting with
       the input layer and ending with the output layer */
    ...);

/* Constructs a backpropagation neural network from a configuration file.
 */
struct fann * fann_create_from_file(const char *configuration_file);

/* Destructs the entire network.
   Be sure to call this function after finished using the network.
 */
void fann_destroy(struct fann *ann);

/* Save the entire network to a configuration file.
 */
void fann_save(struct fann *ann, const char *configuration_file);

/* Saves the entire network to a configuration file.
   But it is saved in fixed point format no matter which
   format it is currently in.

   This is usefull for training a network in floating points,
   and then later executing it in fixed point.

   The function returns the bit position of the fix point, which
   can be used to find out how accurate the fixed point network will be.
   A high value indicates high precision, and a low value indicates low
   precision.

   A negative value indicates very low precision, and a very
   strong possibility for overflow.
   (the actual fix point will be set to 0, since a negative
   fix point does not make sence).

   Generally, a fix point lower than 6 is bad, and should be avoided.
   The best way to avoid this, is to have less connections to each neuron,
   or just less neurons in each layer.

   The fixed point use of this network is only intended for use on machines that
   have no floating point processor, like an iPAQ. On normal computers the floating
   point version is actually faster.
*/
int fann_save_to_fixed(struct fann *ann, const char *configuration_file);
```

```
/* ---- Some stuff to set options on the network on the fly. ---- */

/* Set the learning rate.
 */
void fann_set_learning_rate(struct fann *ann, float learning_rate);

/* The possible activation functions.
   Threshold can not be used, when training the network.
 */
#define FANN_SIGMOID 1
#define FANN_THRESHOLD 2

/* Set the activation function for the hidden layers (default SIGMOID).
 */
void fann_set_activation_function_hidden(struct fann *ann, unsigned int activation_function);

/* Set the activation function for the output layer (default SIGMOID).
 */
void fann_set_activation_function_output(struct fann *ann, unsigned int activation_function);

/* Set the steepness of the sigmoid function used in the hidden layers.
   Only usefull if sigmoid function is used in the hidden layers (default 0.5).
 */
void fann_set_activation_hidden_steepness(struct fann *ann, fann_type steepness);

/* Set the steepness of the sigmoid function used in the output layer.
   Only usefull if sigmoid function is used in the output layer (default 0.5).
 */
void fann_set_activation_output_steepness(struct fann *ann, fann_type steepness);

/* ---- Some stuff to read network options from the network. ---- */

/* Get the learning rate.
 */
float fann_get_learning_rate(struct fann *ann);

/* Get the number of input neurons.
 */
unsigned int fann_get_num_input(struct fann *ann);

/* Get the number of output neurons.
 */
unsigned int fann_get_num_output(struct fann *ann);

/* Get the activation function used in the hidden layers.
 */
unsigned int fann_get_activation_function_hidden(struct fann *ann);

/* Get the activation function used in the output layer.
 */
unsigned int fann_get_activation_function_output(struct fann *ann);

/* Get the steepness parameter for the sigmoid function used in the hidden layers.
 */
fann_type fann_get_activation_hidden_steepness(struct fann *ann);

/* Get the steepness parameter for the sigmoid function used in the output layer.
 */
fann_type fann_get_activation_output_steepness(struct fann *ann);

/* Get the total number of neurons in the entire network.
 */
unsigned int fann_get_total_neurons(struct fann *ann);

/* Get the total number of connections in the entire network.
 */
unsigned int fann_get_total_connections(struct fann *ann);

/* Randomize weights (from the beginning the weights are random between -0.1 and 0.1)
 */
void fann_randomize_weights(struct fann *ann, fann_type min_weight, fann_type max_weight);

/* ---- Training ---- */

#ifndef FIXEDFANN
/* Train one iteration with a set of inputs, and a set of desired outputs.
 */
void fann_train(struct fann *ann, fann_type *input, fann_type *desired_output);
#endif
```

```
/* Test with a set of inputs, and a set of desired outputs.
   This operation updates the mean square error, but does not
   change the network in any way.
*/
fann_type *fann_test(struct fann *ann, fann_type *input, fann_type *desired_output);

/* Reads a file that stores training data, in the format:
   num_train_data num_input num_output\n
   inputdata seperated by space\n
   outputdata seperated by space\n

   .
   .
   .

   inputdata seperated by space\n
   outputdata seperated by space\n
*/
struct fann_train_data* fann_read_train_from_file(char *filename);

/* Destructs the training data
   Be sure to call this function after finished using the training data.
 */
void fann_destroy_train(struct fann_train_data* train_data);

#ifndef FIXEDFANN
/* Trains on an entire dataset, for a maximum of max_epochs
   epochs or until mean square error is lower than desired_error.
   Reports about the progress is given every
   epochs_between_reports epochs.
   If epochs_between_reports is zero, no reports are given.
*/
void fann_train_on_data(struct fann *ann, struct fann_train_data *data, unsigned int max_epochs,
unsigned int epochs_between_reports, float desired_error);

/* Does the same as train_on_data, but reads the data directly from a file.
 */
void fann_train_on_file(struct fann *ann, char *filename, unsigned int max_epochs, unsigned int
epochs_between_reports, float desired_error);
#endif

/* Save the training structure to a file.
 */
void fann_save_train(struct fann_train_data* data, char *filename);

/* Saves the training structure to a fixed point data file.
 *  (Very usefull for testing the quality of a fixed point network).
 */
void fann_save_train_to_fixed(struct fann_train_data* data, char *filename, unsigned int
decimal_point);

/* Reads the mean square error from the network.
 */
float fann_get_error(struct fann *ann);

/* Resets the mean square error from the network.
 */
void fann_reset_error(struct fann *ann);

/* --- Running --- */

/* Runs a input through the network, and returns the output.
 */
fann_type* fann_run(struct fann *ann, fann_type *input);

#ifdef FIXEDFANN

/* returns the position of the decimal point.
 */
unsigned int fann_get_decimal_point(struct fann *ann);

/* returns the multiplier that fix point data is multiplied with.
 */
unsigned int fann_get_multiplier(struct fann *ann);
#endif

#ifdef __cplusplus
}
#endif
```

B.1.2 fann_data.h

```
#ifndef __fann_data_h__
#define __fann_data_h__

/* ---- Data structures ----
 * No data within these structures should be altered directly by the user.
 */

struct fann_neuron
{
    fann_type *weights;
    struct fann_neuron **connected_neurons;
    unsigned int num_connections;
    fann_type value;
}__attribute__((packed));

/* A single layer in the neural network.
 */
struct fann_layer
{
    /* A pointer to the first neuron in the layer
     * When allocated, all the neurons in all the layers are actually
     * in one long array, this is because we wan't to easily clear all
     * the neurons at once.
     */
    struct fann_neuron *first_neuron;

    /* A pointer to the neuron past the last neuron in the layer */
    /* the number of neurons is last_neuron - first_neuron */
    struct fann_neuron *last_neuron;
};

/* The fast artificial neural network(fann) structure
 */
struct fann
{
    /* the learning rate of the network */
    float learning_rate;

    /* the connection rate of the network
     * between 0 and 1, 1 meaning fully connected
     */
    float connection_rate;

    /* pointer to the first layer (input layer) in an array af all the layers,
     * including the input and outputlayers
     */
    struct fann_layer *first_layer;

    /* pointer to the layer past the last layer in an array af all the layers,
     * including the input and outputlayers
     */
    struct fann_layer *last_layer;

    /* Total number of neurons.
     * very usefull, because the actual neurons are allocated in one long array
     */
    unsigned int total_neurons;

    /* Number of input neurons (not calculating bias) */
    unsigned int num_input;

    /* Number of output neurons (not calculating bias) */
    unsigned int num_output;

    /* Used to contain the error deltas used during training
     * Is allocated during first training session,
     * which means that if we do not train, it is never allocated.
     */
    fann_type *train_deltas;

    /* Used to choose which activation function to use

       Sometimes it can be smart, to set the activation function for the hidden neurons
       to THRESHOLD and the activation function for the output neurons to SIGMOID,
       in this way you get a very fast network, that is still cabable of
       producing real valued output.
     */
    unsigned int activation_function_hidden, activation_function_output;
```

```
    /* Parameters for the activation function */
    fann_type activation_hidden_steepness;
    fann_type activation_output_steepness;

#ifdef FIXEDFANN
    /* the decimal_point, used for shifting the fix point
       in fixed point integer operatons.
    */
    unsigned int decimal_point;

    /* the multiplier, used for multiplying the fix point
       in fixed point integer operatons.
       Only used in special cases, since the decimal_point is much faster.
    */
    unsigned int multiplier;

    /* When in fixed point, the sigmoid function is calculated as a stepwise linear
       function. In the activa-
tion_results array, the result is saved, and in the two values arrays,
       the values that gives the results are saved.
     */
    fann_type activation_results[6];
    fann_type activation_hidden_values[6];
    fann_type activation_output_values[6];

#endif

    /* Total number of connections.
     * very usefull, because the actual connections
     * are allocated in one long array
     */
    unsigned int total_connections;

    /* used to store outputs in */
    fann_type *output;

    /* the number of data used to calculate the error.
     */
    unsigned int num_errors;

    /* the total error value.
       the real mean square error is error_value/num_errors
     */
    float error_value;
};

/* Structure used to store data, for use with training. */
struct fann_train_data
{
    unsigned int num_data;
    unsigned int num_input;
    unsigned int num_output;
    fann_type **input;
    fann_type **output;
};

#endif
```

B.1.3 floatfann.h

```
#ifndef __floatfann_h__
#define __floatfann_h__

typedef float fann_type;
#define FLOATFANN
#define FANNPRINTF "%.20e"
#define FANNSCANF "%f"

#include "fann.h"

#endif
```

B.1.4 doublefann.h

```
#ifndef __doublefann_h__
#define __doublefann_h__

typedef double fann_type;
#define DOUBLEFANN
#define FANNPRINTF "%.20e"
#define FANNSCANF "%le"

#include "fann.h"

#endif
```

B.1.5 fixedfann.h

```
#ifndef __fixedfann_h__
#define __fixedfann_h__

typedef int fann_type;
#define FIXEDFANN
#define FANNPRINTF "%d"
#define FANNSCANF "%d"

#include "fann.h"

#endif
```

B.1.6 fann_internal.h

```
#ifndef __fann_internal_h__
#define __fann_internal_h__
/* internal include file, not to be included directly
 */

#include <math.h>
#include "fann_data.h"

#define FANN_FIX_VERSION "FANN_FIX_0.1"
#define FANN_FLO_VERSION "FANN_FLO_0.1"

#ifdef FIXEDFANN
#define FANN_VERSION FANN_FIX_VERSION
#else
#define FANN_VERSION FANN_FLO_VERSION
#endif

struct fann * fann_allocate_structure(float learning_rate, unsigned int num_layers);
void fann_allocate_neurons(struct fann *ann);

void fann_allocate_connections(struct fann *ann);

int fann_save_internal(struct fann *ann, const char *configuration_file, unsigned int save_as_fixed);
void fann_save_train_internal(struct fann_train_data* data, char *filename, unsigned int
save_as_fixed, unsigned int decimal_point);

int fann_compare_connections(const void* c1, const void* c2);
void fann_seed_rand();

/* called fann_max, in order to not interferre with predefined versions of max */
#define fann_max(x, y) (((x) > (y)) ? (x) : (y))
#define fann_min(x, y) (((x) < (y)) ? (x) : (y))

#define fann_rand(min_value, max_value)
(((double)(min_value))+(((double)(max_value)-((double)(min_value)))*rand()/(RAND_MAX+1.0)))

#define fann_abs(value) (((value) > 0) ? (value) : -(value))

#ifdef FIXEDFANN

#define fann_mult(x,y) ((x*y) >> decimal_point)
#define fann_div(x,y) (((x) << decimal_point)/y)
#define fann_random_weight() (fann_type)(fann_rand(-multiplier/10,multiplier/10))
/* sigmoid calculated with use of floats, only as reference */
#define fann_sigmoid(steepness, value) ((fann_type)(0.5+((1.0/(1.0 + exp(-2.0 *
((float)steepness/multiplier) * ((float)value/multiplier))))*multiplier)))
/* sigmoid as a stepwise linear function */
#define fann_linear(v1, r1, v2, r2, value) ((((r2-r1) * (value-v1))/(v2-v1)) + r1)
#define fann_sigmoid_stepwise(v1, v2, v3, v4, v5, v6, r1, r2, r3, r4, r5, r6, value, multiplier) (value < v5
? (value < v3 ? (value < v2 ? (value < v1 ? 0 : fann_linear(v1, r1, v2, r2, value)) : fann_linear(v2, r2,
v3, r3, value)) : (value < v4 ? fann_linear(v3, r3, v4, r4, value) : fann_linear(v4, r4, v5, r5, value))) :
(value < v6 ? fann_linear(v5, r5, v6, r6, value) : multiplier))
#else

#define fann_mult(x,y) (x*y)
#define fann_div(x,y) (x/y)
#define fann_random_weight() (fann_rand(-0.1,0.1))
#define fann_sigmoid(steepness, value) (1.0/(1.0 + exp(-2.0 * steepness * value)))
#define fann_sigmoid_derive(steepness, value) (2.0 * steepness * value * (1.0 - value))

#endif

#endif
```

B.1.7 fann.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>

/* create a neural network.
 */
struct fann * fann_create(float connection_rate, float learning_rate,
    unsigned int num_layers,   /* the number of layers, including the input and output layer */
    ...)
/* the number of neurons in each of the layers, starting with the input layer and ending with the output layer */
{
    va_list layer_sizes;
    struct fann_layer *layer_it, *last_layer, *prev_layer;
    struct fann *ann;
    struct fann_neuron *neuron_it, *last_neuron, *random_neuron, *bias_neuron;
    unsigned int prev_layer_size, i, j;
    unsigned int num_neurons_in, num_neurons_out;
    unsigned int min_connections, max_connections, num_connections;
    unsigned int connections_per_neuron, allocated_connections;
    unsigned int random_number, found_connection;

#ifdef FIXEDFANN
    unsigned int decimal_point;
    unsigned int multiplier;
#endif

    if(connection_rate > 1){
        connection_rate = 1;
    }

    /* seed random */
    fann_seed_rand();

    /* allocate the general structure */
    ann = fann_allocate_structure(learning_rate, num_layers);
    ann->connection_rate = connection_rate;
#ifdef FIXEDFANN
    decimal_point = ann->decimal_point;
    multiplier = ann->multiplier;
#endif

    /* determine how many neurons there should be in each layer */
    va_start(layer_sizes, num_layers);
    for(layer_it = ann->first_layer; layer_it != ann->last_layer; layer_it++){
        /* we do not allocate room here, but we make sure that
           last_neuron - first_neuron is the number of neurons */
        layer_it->first_neuron = NULL;
        layer_it->last_neuron = layer_it->first_neuron + va_arg(layer_sizes, unsigned int) +1;
/* +1 for bias */

        ann->total_neurons += layer_it->last_neuron - layer_it->first_neuron;
    }
    va_end(layer_sizes);

    ann->num_output = (ann->last_layer-1)->last_neuron - (ann->last_layer-1)->first_neuron -1;
    ann->num_input = ann->first_layer->last_neuron - ann->first_layer->first_neuron -1;

    /* allocate room for the actual neurons */
    fann_allocate_neurons(ann);

#ifdef DEBUG
    printf("creating network with learning rate %f and connection rate %f\n", learning_rate,
connection_rate);
    printf("input\n");
    printf(" layer :  %d neurons, 1 bias\n", ann->first_layer->last_neuron -
ann->first_layer->first_neuron - 1);
#endif

    num_neurons_in = ann->num_input;
    for(layer_it = ann->first_layer+1; layer_it != ann->last_layer; layer_it++){
        num_neurons_out = layer_it->last_neuron - layer_it->first_neuron - 1;
        /* if all neurons in each layer should be connected to at least one neuron
           in the previous layer, and one neuron in the next layer.
           and the bias node should be connected to the all neurons in the next layer.
           Then this is the minimum amount of neurons */
        min_connections = fann_max(num_neurons_in, num_neurons_out) + num_neurons_out;
        max_connections = num_neurons_in * num_neurons_out;            /* not calculating bias */
```

```
        num_connections = fann_max(min_connections,
            (unsigned int)(0.5+(connection_rate * max_connections)) + num_neurons_out);

        ann->total_connections += num_connections;

        connections_per_neuron = num_connections/num_neurons_out;
        allocated_connections = 0;
        /* Now split out the connections on the different neurons */
        for(i = 0; i != num_neurons_out; i++){
            layer_it->first_neuron[i].num_connections = connections_per_neuron;
            allocated_connections += connections_per_neuron;

            if(allocated_connections < (num_connections*(i+1))/num_neurons_out){
                layer_it->first_neuron[i].num_connections++;
                allocated_connections++;
            }
        }

        /* used in the next run of the loop */
        num_neurons_in = num_neurons_out;
    }

    fann_allocate_connections(ann);

    if(connection_rate == 1){
        prev_layer_size = ann->num_input+1;
        prev_layer = ann->first_layer;
        last_layer = ann->last_layer;
        for(layer_it = ann->first_layer+1; layer_it != last_layer; layer_it++){
            last_neuron = layer_it->last_neuron-1;
            for(neuron_it = layer_it->first_neuron; neuron_it != last_neuron; neuron_it++){
                for(i = 0; i != prev_layer_size; i++){
                    neuron_it->weights[i] = fann_random_weight();
                    /* these connections are still initialized for fully connected networks, to allow
                       operations to work, that are not optimized for fully connected networks.
                    */
                    neuron_it->connected_neurons[i] = prev_layer->first_neuron+i;
                }

            }
            prev_layer_size = layer_it->last_neuron - layer_it->first_neuron;
            prev_layer = layer_it;
#ifdef DEBUG
            printf(" layer :  %d neurons, 1 bias\n", prev_layer_size-1);
#endif
        }
    }else{
        /* make connections for a network, that are not fully connected */

        /* generally, what we do is first to connect all the input
           neurons to a output neuron, respecting the number of
           available input neurons for each output neuron. Then
           we go through all the output neurons, and connect the
           rest of the connections to input neurons, that they are
           not allready connected to.
        */

        /* first clear all the connections, because we want to
           be able to see which connections are allready connected */
        memset((ann->first_layer+1)->first_neuron->connected_neurons, 0, ann->total_connections *
sizeof(struct fann_neuron*));

        for(layer_it = ann->first_layer+1;
            layer_it != ann->last_layer; layer_it++){

            num_neurons_out = layer_it->last_neuron - layer_it->first_neuron - 1;
            num_neurons_in = (layer_it-1)->last_neuron - (layer_it-1)->first_neuron - 1;

            /* first connect the bias neuron */
            bias_neuron = (layer_it-1)->last_neuron-1;
            last_neuron = layer_it->last_neuron-1;
            for(neuron_it = layer_it->first_neuron;
                neuron_it != last_neuron; neuron_it++){

                neuron_it->connected_neurons[0] = bias_neuron;
                neuron_it->weights[0] = fann_random_weight();
            }

            /* then connect all neurons in the input layer */
            last_neuron = (layer_it-1)->last_neuron - 1;
```

```
            for(neuron_it = (layer_it-1)→first_neuron;
                neuron_it ≠ last_neuron; neuron_it++){

                /* random neuron in the output layer that has space
                   for more connections */
                do {
                    random_number = (int) (0.5+fann_rand(0, num_neurons_out-1));
                    random_neuron = layer_it→first_neuron + random_number;
                    /* checks the last space in the connections array for room */
                }while(random_neuron→connected_neurons[random_neuron→num_connections-1]);

                /* find an empty space in the connection array and connect */
                for(i = 0; i < random_neuron→num_connections; i++){
                    if(random_neuron→connected_neurons[i] == NULL){
                        random_neuron→connected_neurons[i] = neuron_it;
                        random_neuron→weights[i] = fann_random_weight();
                        break;
                    }
                }
            }

            /* then connect the rest of the unconnected neurons */
            last_neuron = layer_it→last_neuron - 1;
            for(neuron_it = layer_it→first_neuron;
                neuron_it ≠ last_neuron; neuron_it++){
                /* find empty space in the connection array and connect */
                for(i = 0; i < neuron_it→num_connections; i++){
                    /* continue if allready connected */
                    if(neuron_it→connected_neurons[i] ≠ NULL) continue;

                    do {
                        found_connection = 0;
                        random_number = (int) (0.5+fann_rand(0, num_neurons_in-1));
                        random_neuron = (layer_it-1)→first_neuron + random_number;

                        /* check to see if this connection is allready there */
                        for(j = 0; j < i; j++){
                            if(random_neuron == neuron_it→connected_neurons[j]){
                                found_connection = 1;
                                break;
                            }
                        }

                    }while(found_connection);

                    /* we have found a neuron that is not allready
                       connected to us, connect it */
                    neuron_it→connected_neurons[i] = random_neuron;
                    neuron_it→weights[i] = fann_random_weight();
                }
            }

#ifdef DEBUG
            printf(" layer :  %d neurons, 1 bias\n", num_neurons_out);
#endif
        }

        /* TODO it would be nice to have the randomly created connections sorted
           for smoother memory access.
        */
    }

#ifdef DEBUG
    printf("output\n");
#endif

    return ann;
}


/* Create a network from a configuration file.
 */
struct fann * fann_create_from_file(const char *configuration_file)
{
    unsigned int num_layers, layer_size, activation_function_hidden, activation_function_output,
input_neuron, i;
#ifdef FIXEDFANN
    unsigned int decimal_point, multiplier;
#endif
    fann_type activation_hidden_steepness, activation_output_steepness;
```

```
    float learning_rate, connection_rate;
    struct fann_neuron *first_neuron, *neuron_it, *last_neuron, **connected_neurons;
    fann_type *weights;
    struct fann_layer *layer_it;
    struct fann *ann;

    char *read_version;
    FILE *conf = fopen(configuration_file, "r");

    if(!conf){
        printf("Unable to open configuration file \"%s\" for reading.\n", configuration_file);
        return NULL;
    }

    read_version = (char *)calloc(strlen(FANN_VERSION"\n"), 1);
    fread(read_version, 1, strlen(FANN_VERSION"\n"), conf);          /* reads version */

    /* compares the version information */
    if(strncmp(read_version, FANN_VERSION"\n", strlen(FANN_VERSION"\n")) != 0){
        printf("Wrong version, aborting read of configuration file \"%s\".\n", configuration_file);
        return NULL;
    }

#ifdef FIXEDFANN
    if(fscanf(conf, "%u\n", &decimal_point) != 1){
        printf("Error reading info from configuration file \"%s\".\n", configuration_file);
        return NULL;
    }
    multiplier = 1 << decimal_point;
#endif

    if(fscanf(conf, "%u %f %f %u %u "FANNSCANF" "FANNSCANF"\n", &num_layers,
&learning_rate, &connection_rate, &activation_function_hidden, &activation_function_output,
&activation_hidden_steepness, &activation_output_steepness) != 7){
        printf("Error reading info from configuration file \"%s\".\n", configuration_file);
        return NULL;
    }

    ann = fann_allocate_structure(learning_rate, num_layers);
#ifdef FIXEDFANN
    ann->decimal_point = decimal_point;
    ann->multiplier = multiplier;
    ann->activation_function_hidden = activation_function_hidden;
    ann->activation_function_output = activation_function_output;
    ann->activation_hidden_steepness = activation_hidden_steepness;
    ann->activation_output_steepness = activation_output_steepness;
    ann->connection_rate = connection_rate;

    /* Calculate the parameters for the stepwise linear sigmoid function fixed point.
       Using a rewritten sigmoid function.
       results 0.005, 0.05, 0.25, 0.75, 0.95, 0.995
     */
    ann->activation_results[0] = (fann_type)(multiplier/200.0+0.5);
    ann->activation_results[1] = (fann_type)(multiplier/20.0+0.5);
    ann->activation_results[2] = (fann_type)(multiplier/4.0+0.5);
    ann->activation_results[3] = multiplier - (fann_type)(multiplier/4.0+0.5);
    ann->activation_results[4] = multiplier - (fann_type)(multiplier/20.0+0.5);
    ann->activation_results[5] = multiplier - (fann_type)(multiplier/200.0+0.5);

    fann_set_activation_hidden_steepness(ann, activation_hidden_steepness);
    fann_set_activation_output_steepness(ann, activation_output_steepness);
#endif

#ifdef DEBUG
    printf("creating network with learning rate %f\n", learning_rate);
    printf("input\n");
#endif

    /* determine how many neurons there should be in each layer */
    for(layer_it = ann->first_layer; layer_it != ann->last_layer; layer_it++){
        if(fscanf(conf, "%u ", &layer_size) != 1){
            printf("Error reading neuron info from configuration file \"%s\".\n",
configuration_file);
            return ann;
        }
        /* we do not allocate room here, but we make sure that
           last_neuron - first_neuron is the number of neurons */
        layer_it->first_neuron = NULL;
        layer_it->last_neuron = layer_it->first_neuron + layer_size;
        ann->total_neurons += layer_size;
```

```
#ifdef DEBUG
        printf(" layer :  %d neurons, 1 bias\n", layer_size);
#endif
    }

    ann→num_input = ann→first_layer→last_neuron - ann→first_layer→first_neuron;
    ann→num_output = ((ann→last_layer-1)→last_neuron - (ann→last_layer-1)→first_neuron) - 1;

    /* allocate room for the actual neurons */
    fann_allocate_neurons(ann);

    last_neuron = (ann→last_layer-1)→last_neuron;
    for(neuron_it = ann→first_layer→first_neuron;
        neuron_it ≠ last_neuron; neuron_it++){
        if(fscanf(conf, "%u ", &neuron_it→num_connections) ≠ 1){
            printf("Error reading neuron info from configuration file \"%s\".\n",
configuration_file);
            return ann;
        }
        ann→total_connections += neuron_it→num_connections;
    }

    fann_allocate_connections(ann);

    connected_neurons = (ann→first_layer+1)→first_neuron→connected_neurons;
    weights = (ann→first_layer+1)→first_neuron→weights;
    first_neuron = ann→first_layer→first_neuron;

    for(i = 0; i < ann→total_connections; i++){
        if(fscanf(conf, "(%u "FANNSCANF") ", &input_neuron, &weights[i]) ≠ 2){
            printf("Error reading connections from configuration file \"%s\".\n",
configuration_file);
            return ann;
        }
        connected_neurons[i] = first_neuron+input_neuron;
    }

#ifdef DEBUG
    printf("output\n");
#endif
    fclose(conf);
    return ann;
}


/* deallocate the network.
 */
void fann_destroy(struct fann *ann)
{
    free((ann→first_layer+1)→first_neuron→weights);
    free((ann→first_layer+1)→first_neuron→connected_neurons);
    free(ann→first_layer→first_neuron);
    free(ann→first_layer);
    free(ann→output);
    if(ann→train_deltas ≠ NULL) free(ann→train_deltas);
    free(ann);
}

/* Save the network.
 */
void fann_save(struct fann *ann, const char *configuration_file)
{
    fann_save_internal(ann, configuration_file, 0);
}

/* Save the network as fixed point data.
 */
int fann_save_to_fixed(struct fann *ann, const char *configuration_file)
{
    return fann_save_internal(ann, configuration_file, 1);
}

void fann_set_learning_rate(struct fann *ann, float learning_rate)
{
    ann→learning_rate = learning_rate;
}

void fann_set_activation_function_hidden(struct fann *ann, unsigned int activation_function)
{
    ann→activation_function_hidden = activation_function;
```

```
}

void fann_set_activation_function_output(struct fann *ann, unsigned int activation_function)
{
    ann->activation_function_output = activation_function;
}

void fann_set_activation_hidden_steepness(struct fann *ann, fann_type steepness)
{
#ifdef FIXEDFANN
    int i;
#endif
    ann->activation_hidden_steepness = steepness;
#ifdef FIXEDFANN
    for(i = 0; i < 6; i++){
        ann->activation_hidden_values[i] =
(fann_type)((((log(ann->multiplier/(float)ann->activation_results[i] -1)*(float)ann->multiplier) /
-2.0)*(float)ann->multiplier) / steepness);
    }
#endif
}

void fann_set_activation_output_steepness(struct fann *ann, fann_type steepness)
{
#ifdef FIXEDFANN
    int i;
#endif
    ann->activation_hidden_steepness = steepness;
#ifdef FIXEDFANN
    for(i = 0; i < 6; i++){
        ann->activation_output_values[i] =
(fann_type)((((log(ann->multiplier/(float)ann->activation_results[i] -1)*(float)ann->multiplier) /
-2.0)*(float)ann->multiplier) / steepness);
    }
#endif
}

float fann_get_learning_rate(struct fann *ann)
{
    return ann->learning_rate;
}

unsigned int fann_get_num_input(struct fann *ann)
{
    return ann->num_input;
}

unsigned int fann_get_num_output(struct fann *ann)
{
    return ann->num_output;
}

unsigned int fann_get_activation_function_hidden(struct fann *ann)
{
    return ann->activation_function_hidden;
}

unsigned int fann_get_activation_function_output(struct fann *ann)
{
    return ann->activation_function_output;
}

fann_type fann_get_activation_hidden_steepness(struct fann *ann)
{
    return ann->activation_hidden_steepness;
}

fann_type fann_get_activation_output_steepness(struct fann *ann)
{
    return ann->activation_output_steepness;
}

unsigned int fann_get_total_neurons(struct fann *ann)
{
    /* -1, because there is always an unused bias neuron in the last layer */
    return ann->total_neurons - 1;
}

unsigned int fann_get_total_connections(struct fann *ann)
{
```

```
    return ann→total_connections;
}

void fann_randomize_weights(struct fann *ann, fann_type min_weight, fann_type max_weight)
{
    fann_type *last_weight;
    fann_type *weights = (ann→first_layer+1)→first_neuron→weights;
    last_weight = weights + ann→total_connections;
    for(;weights ≠ last_weight; weights++){
        *weights = (fann_type)(fann_rand(min_weight, max_weight));
    }
}

#ifndef FIXEDFANN
/* Trains the network with the backpropagation algorithm.
 */
void fann_train(struct fann *ann, fann_type *input, fann_type *desired_output)
{
    struct fann_neuron *neuron_it, *last_neuron, *neurons;
    fann_type neuron_value, *delta_it, *delta_begin, tmp_delta;
    struct fann_layer *layer_it;
    unsigned int i, shift_prev_layer;

    /* store some variabels local for fast access */
    const float learning_rate = ann→learning_rate;
    const fann_type activation_output_steepness = ann→activation_output_steepness;
    const fann_type activation_hidden_steepness = ann→activation_hidden_steepness;
    const struct fann_neuron *first_neuron = ann→first_layer→first_neuron;

    const struct fann_neuron *last_layer_begin = (ann→last_layer-1)→first_neuron;
    const struct fann_neuron *last_layer_end = last_layer_begin + ann→num_output;
    struct fann_layer *first_layer = ann→first_layer;
    struct fann_layer *last_layer = ann→last_layer;

    fann_run(ann, input);
    /* if no room allocated for the delta variabels, allocate it now */
    if(ann→train_deltas == NULL){
        ann→train_deltas = (fann_type *)calloc(ann→total_neurons, sizeof(fann_type));
    }
    delta_begin = ann→train_deltas;

    /* clear the delta variabels */
    memset(delta_begin, 0, (ann→total_neurons) * sizeof(fann_type));

#ifdef DEBUGTRAIN
    printf("calculate deltas\n");
#endif
    /* calculate the error and place it in the output layer */
    delta_it = delta_begin + (last_layer_begin - first_neuron);
    for(; last_layer_begin ≠ last_layer_end; last_layer_begin++){
        neuron_value = last_layer_begin→value;
        /* TODO add switch the minute there are other activation functions */
        *delta_it = fann_sigmoid_derive(activation_output_steepness, neuron_value) * (*desired_output
- neuron_value);

        ann→error_value += (*desired_output - neuron_value) * (*desired_output - neuron_value);

#ifdef DEBUGTRAIN
        printf("delta[%d] = "FANNPRINTF"\n", (delta_it - delta_begin), *delta_it);
#endif
        desired_output++;
        delta_it++;
    }
    ann→num_errors++;


    /* go through all the layers, from last to first. And propagate the error backwards */
    for(layer_it = last_layer-1; layer_it ≠ first_layer; --layer_it){
        last_neuron = layer_it→last_neuron;

        /* for each connection in this layer, propagate the error backwards*/
        if(ann→connection_rate == 1){           /* optimization for fully connected networks */
            shift_prev_layer = (layer_it-1)→first_neuron - first_neuron;
            for(neuron_it = layer_it→first_neuron;
                neuron_it ≠ last_neuron; neuron_it++){
                tmp_delta = *(delta_begin + (neuron_it - first_neuron));
                for(i = 0; i < neuron_it→num_connections; i++){
                    *(delta_begin + i + shift_prev_layer) += tmp_delta * neuron_it→weights[i];
                }
            }
```

```
        }else{
            for(neuron_it = layer_it→first_neuron;
                neuron_it ≠ last_neuron; neuron_it++){
                tmp_delta = ∗(delta_begin + (neuron_it - first_neuron));
                for(i = 0; i < neuron_it→num_connections; i++){
                    ∗(delta_begin + (neuron_it→connected_neurons[i] - first_neuron)) +=
                        tmp_delta ∗ neuron_it→weights[i];
                }
            }
        }

        /∗ then calculate the actual errors in the previous layer ∗/
        delta_it = delta_begin + ((layer_it-1)→first_neuron - first_neuron);
        last_neuron = (layer_it-1)→last_neuron;
        for(neuron_it = (layer_it-1)→first_neuron;
            neuron_it ≠ last_neuron; neuron_it++){
            neuron_value = neuron_it→value;
            /∗ TODO add switch the minute there are other activation functions ∗/
            ∗delta_it ∗ = fann_sigmoid_derive(activation_hidden_steepness, neuron_value) ∗
learning_rate;

#ifdef DEBUGTRAIN
            printf("delta[%d] = "FANNPRINTF"\n", delta_it - delta_begin, ∗delta_it);
#endif
            delta_it++;
        }
    }

#ifdef DEBUGTRAIN
    printf("\nupdate weights\n");
#endif

    for(layer_it = (first_layer+1); layer_it ≠ last_layer; layer_it++){
#ifdef DEBUGTRAIN
        printf("layer[%d]\n", layer_it - first_layer);
#endif
        last_neuron = layer_it→last_neuron;
        if(ann→connection_rate == 1){           /∗ optimization for fully connected networks ∗/
            neurons = (layer_it-1)→first_neuron;
            for(neuron_it = layer_it→first_neuron;
                neuron_it ≠ last_neuron; neuron_it++){
                tmp_delta = ∗(delta_begin + (neuron_it - first_neuron));
                for(i = 0; i < neuron_it→num_connections; i++){
                    neuron_it→weights[i] += tmp_delta ∗ neurons[i].value;
                }
            }
        }else{
            for(neuron_it = layer_it→first_neuron;
                neuron_it ≠ last_neuron; neuron_it++){
                tmp_delta = ∗(delta_begin + (neuron_it - first_neuron));
                for(i = 0; i < neuron_it→num_connections; i++){
                    neuron_it→weights[i] += tmp_delta ∗ neuron_it→connected_neurons[i]→value;
                }
            }
        }
    }
}
#endif

/∗ Tests the network.
 ∗/
fann_type ∗fann_test(struct fann ∗ann, fann_type ∗input, fann_type ∗desired_output)
{
    fann_type neuron_value;
    fann_type ∗output_begin = fann_run(ann, input);
    fann_type ∗output_it;
    const fann_type ∗output_end = output_begin + ann→num_output;

    /∗ calculate the error ∗/
    for(output_it = output_begin;
        output_it ≠ output_end; output_it++){
        neuron_value = ∗output_it;

#ifdef FIXEDFANN
        ann→error_value += ((∗desired_output - neuron_value)/(float)ann→multiplier) ∗
((∗desired_output - neuron_value)/(float)ann→multiplier);
#else
        ann→error_value += (∗desired_output - neuron_value) ∗ (∗desired_output - neuron_value);
#endif
```

```
            desired_output++;
        }
        ann->num_errors++;

        return output_begin;
}

/* Reads training data from a file.
 */
struct fann_train_data* fann_read_train_from_file(char *filename)
{
        unsigned int num_input, num_output, num_data, i, j;
        unsigned int line = 1;
        struct fann_train_data* data;

        FILE *file = fopen(filename, "r");

        data = (struct fann_train_data *)malloc(sizeof(struct fann_train_data));

        if(!file){
            printf("Unable to open train data file \"%s\" for reading.\n", filename);
            return NULL;
        }

        if(fscanf(file, "%u %u %u\n", &num_data, &num_input, &num_output) != 3){
            printf("Error reading info from train data file \"%s\", line:  %d.\n", filename, line);
            return NULL;
        }
        line++;

        data->num_data = num_data;
        data->num_input = num_input;
        data->num_output = num_output;
        data->input = (fann_type **)calloc(num_data, sizeof(fann_type *));
        data->output = (fann_type **)calloc(num_data, sizeof(fann_type *));

        for(i = 0; i != num_data; i++){
            data->input[i] = (fann_type *)calloc(num_input, sizeof(fann_type));
            for(j = 0; j != num_input; j++){
                if(fscanf(file, FANNSCANF" ", &data->input[i][j]) != 1){
                    printf("Error reading info from train data file \"%s\", line:  %d.\n", filename,
line);
                    return NULL;
                }
            }
            line++;

            data->output[i] = (fann_type *)calloc(num_output, sizeof(fann_type));
            for(j = 0; j != num_output; j++){
                if(fscanf(file, FANNSCANF" ", &data->output[i][j]) != 1){
                    printf("Error reading info from train data file \"%s\", line:  %d.\n", filename,
line);
                    return NULL;
                }
            }
            line++;
        }

        return data;
}

/* Save training data to a file
 */
void fann_save_train(struct fann_train_data* data, char *filename)
{
        fann_save_train_internal(data, filename, 0, 0);
}

/* Save training data to a file in fixed point algebra.
    (Good for testing a network in fixed point)
 */
void fann_save_train_to_fixed(struct fann_train_data* data, char *filename, unsigned int
decimal_point)
{
        fann_save_train_internal(data, filename, 1, decimal_point);
}

/* deallocate the train data structure.
 */
void fann_destroy_train(struct fann_train_data *data)
```

```
{
    unsigned int i;
    for(i = 0; i ≠ data→num_data; i++){
        free(data→input[i]);
        free(data→output[i]);
    }
    free(data→input);
    free(data→output);
    free(data);
}

#ifndef FIXEDFANN
/* Train directly on the training data.
 */
void fann_train_on_data(struct fann *ann, struct fann_train_data *data, unsigned int max_epochs,
unsigned int epochs_between_reports, float desired_error)
{
    float error;
    unsigned int i, j;

    if(epochs_between_reports){
        printf("Max epochs %8d.  Desired error:  %.10f\n", max_epochs, desired_error);
    }

    for(i = 1; i ≤ max_epochs; i++){
        /* train */
        fann_reset_error(ann);

        for(j = 0; j ≠ data→num_data; j++){
            fann_train(ann, data→input[j], data→output[j]);
        }

        error = fann_get_error(ann);

        /* print current output */
        if(epochs_between_reports &&
            (i % epochs_between_reports == 0
                || i == max_epochs
                || i == 1
                || error < desired_error)){
            printf("Epochs %8d.  Current error:  %.10f\n", i, error);
        }

        if(error < desired_error){
            break;
        }
    }
    fann_reset_error(ann);
}

/* Wrapper to make it easy to train directly on a training data file.
 */
void fann_train_on_file(struct fann *ann, char *filename, unsigned int max_epochs, unsigned int
epochs_between_reports, float desired_error)
{
    struct fann_train_data *data = fann_read_train_from_file(filename);
    fann_train_on_data(ann, data, max_epochs, epochs_between_reports, desired_error);
    fann_destroy_train(data);
}
#endif

/* get the mean square error.
 */
float fann_get_error(struct fann *ann)
{
    if(ann→num_errors){
        return ann→error_value/(float)ann→num_errors;
    }else{
        return 0;
    }
}

/* reset the mean square error.
 */
void fann_reset_error(struct fann *ann)
{
    ann→num_errors = 0;
    ann→error_value = 0;
}
```

```
#ifdef FIXEDFANN
/* returns the position of the fix point.
 */
unsigned int fann_get_decimal_point(struct fann *ann)
{
    return ann->decimal_point;
}

/* returns the multiplier that fix point data is multiplied with.
 */
unsigned int fann_get_multiplier(struct fann *ann)
{
    return ann->multiplier;
}

#endif

/* runs the network.
 */
fann_type* fann_run(struct fann *ann, fann_type *input)
{
    struct fann_neuron *neuron_it, *last_neuron, *neurons, **neuron_pointers;
    unsigned int activation_function, i, num_connections, num_input, num_output;
    fann_type neuron_value, *weights, *output;
    struct fann_layer *layer_it, *last_layer;


    /* store some variabels local for fast access */
#ifndef FIXEDFANN
    fann_type steepness;
    const fann_type activation_output_steepness = ann->activation_output_steepness;
    const fann_type activation_hidden_steepness = ann->activation_hidden_steepness;
#endif

    unsigned int activation_function_output = ann->activation_function_output;
    unsigned int activation_function_hidden = ann->activation_function_hidden;
    struct fann_neuron *first_neuron = ann->first_layer->first_neuron;
#ifdef FIXEDFANN
    unsigned int multiplier = ann->multiplier;
    unsigned int decimal_point = ann->decimal_point;

    /* values used for the stepwise linear sigmoid function */

    /* the results */
    fann_type r1 = ann->activation_results[0];
    fann_type r2 = ann->activation_results[1];
    fann_type r3 = ann->activation_results[2];
    fann_type r4 = ann->activation_results[3];
    fann_type r5 = ann->activation_results[4];
    fann_type r6 = ann->activation_results[5];

    /* the hidden parameters */
    fann_type h1 = ann->activation_hidden_values[0];
    fann_type h2 = ann->activation_hidden_values[1];
    fann_type h3 = ann->activation_hidden_values[2];
    fann_type h4 = ann->activation_hidden_values[3];
    fann_type h5 = ann->activation_hidden_values[4];
    fann_type h6 = ann->activation_hidden_values[5];

    /* the output parameters */
    fann_type o1 = ann->activation_output_values[0];
    fann_type o2 = ann->activation_output_values[1];
    fann_type o3 = ann->activation_output_values[2];
    fann_type o4 = ann->activation_output_values[3];
    fann_type o5 = ann->activation_output_values[4];
    fann_type o6 = ann->activation_output_values[5];
#endif

    /* first set the input */
    num_input = ann->num_input;
    for(i = 0; i != num_input; i++){
#ifdef FIXEDFANN
        if(fann_abs(input[i]) > multiplier){
            printf("Warning input number %d is out of range -%d - %d with value %d, integer
overflow may occur.\n", i, multiplier, multiplier, input[i]);
        }
#endif
        first_neuron[i].value = input[i];
    }
```

```
    last_layer = ann→last_layer;
    for(layer_it = ann→first_layer+1; layer_it ≠ last_layer; layer_it++){

#ifdef FIXEDFANN
        ((layer_it-1)→last_neuron-1)→value = multiplier;
#else
        /* set the bias neuron */
        ((layer_it-1)→last_neuron-1)→value = 1;

        steepness = (layer_it == last_layer-1) ?
            activation_output_steepness : activation_hidden_steepness;
#endif

        activation_function = (layer_it == last_layer-1) ?
            activation_function_output : activation_function_hidden;

        last_neuron = layer_it→last_neuron-1;
        for(neuron_it = layer_it→first_neuron; neuron_it ≠ last_neuron; neuron_it++){
            neuron_value = 0;
            num_connections = neuron_it→num_connections;
            weights = neuron_it→weights;
            if(ann→connection_rate == 1){
                neurons = (layer_it-1)→first_neuron;

                i = num_connections & 3;                    /* same as modulo 4 */
                switch(i) {
                    case 3:
                        neuron_value += fann_mult(weights[2], neurons[2].value);
                    case 2:
                        neuron_value += fann_mult(weights[1], neurons[1].value);
                    case 1:
                        neuron_value += fann_mult(weights[0], neurons[0].value);
                    case 0:
                        break;
                }

                for(;i ≠ num_connections; i += 4){
                    neuron_value +=
                        fann_mult(weights[i], neurons[i].value) +
                        fann_mult(weights[i+1], neurons[i+1].value) +
                        fann_mult(weights[i+2], neurons[i+2].value) +
                        fann_mult(weights[i+3], neurons[i+3].value);
                }
            }else{
                neuron_pointers = neuron_it→connected_neurons;

                i = num_connections & 3;                    /* same as modulo 4 */
                switch(i) {
                    case 3:
                        neuron_value += fann_mult(weights[2], neuron_pointers[2]→value);
                    case 2:
                        neuron_value += fann_mult(weights[1], neuron_pointers[1]→value);
                    case 1:
                        neuron_value += fann_mult(weights[0], neuron_pointers[0]→value);
                    case 0:
                        break;
                }

                for(;i ≠ num_connections; i += 4){
                    neuron_value +=
                        fann_mult(weights[i], neuron_pointers[i]→value) +
                        fann_mult(weights[i+1], neuron_pointers[i+1]→value) +
                        fann_mult(weights[i+2], neuron_pointers[i+2]→value) +
                        fann_mult(weights[i+3], neuron_pointers[i+3]→value);
                }
            }

            if(activation_function == FANN_SIGMOID){
#ifdef FIXEDFANN
                if(layer_it == last_layer-1){
                    neuron_it→value = fann_sigmoid_stepwise(o1, o2, o3, o4, o5, o6, r1, r2, r3, r4,
r5, r6, neuron_value, multiplier);
                }else{
                    neuron_it→value = fann_sigmoid_stepwise(h1, h2, h3, h4, h5, h6, r1, r2, r3, r4,
r5, r6, neuron_value, multiplier);
                }
#else
                neuron_it→value = fann_sigmoid(steepness, neuron_value);
#endif
            }else{
```

```
                    neuron_it→value = (neuron_value < 0) ? 0 : 1;
                }
            }
        }

        /* set the output */
        output = ann→output;
        num_output = ann→num_output;
        neurons = (ann→last_layer-1)→first_neuron;
        for(i = 0; i ≠ num_output; i++){
            output[i] = neurons[i].value;
        }
        return ann→output;
}
```

B.1.8 fann_internal.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>
#include <time.h>

#include "fann_internal.h"

/* Allocates the main structure and sets some default values.
 */
struct fann * fann_allocate_structure(float learning_rate, unsigned int num_layers)
{
    struct fann *ann;

    if(num_layers < 2){
#ifdef DEBUG
        printf("less than 2 layers - ABORTING.\n");
#endif
        return NULL;
    }

    /* allocate and initialize the main network structure */
    ann = (struct fann *)malloc(sizeof(struct fann));
    ann->learning_rate = learning_rate;
    ann->total_neurons = 0;
    ann->total_connections = 0;
    ann->num_input = 0;
    ann->num_output = 0;
    ann->train_deltas = NULL;
    ann->num_errors = 0;
    ann->error_value = 0;

#ifdef FIXEDFANN
    /* these values are only boring defaults, and should really
       never be used, since the real values are always loaded from a file. */
    ann->decimal_point = 8;
    ann->multiplier = 256;
#endif

    ann->activation_function_hidden = FANN_SIGMOID;
    ann->activation_function_output = FANN_SIGMOID;
#ifdef FIXEDFANN
    ann->activation_hidden_steepness = ann->multiplier/2;
    ann->activation_output_steepness = ann->multiplier/2;
#else
    ann->activation_hidden_steepness = 0.5;
    ann->activation_output_steepness = 0.5;
#endif

    /* allocate room for the layers */
    ann->first_layer = (struct fann_layer *)calloc(num_layers, sizeof(struct fann_layer));
    ann->last_layer = ann->first_layer + num_layers;

    return ann;
}

/* Allocates room for the neurons.
 */
void fann_allocate_neurons(struct fann *ann)
{
    struct fann_layer *layer_it;
    struct fann_neuron *neurons;
    unsigned int num_neurons_so_far = 0;
    unsigned int num_neurons = 0;

    /* all the neurons is allocated in one long array */
    neurons = (struct fann_neuron *)calloc(ann->total_neurons, sizeof(struct fann_neuron));

    /* clear data, primarily to make the input neurons cleared */
    memset(neurons, 0, ann->total_neurons * sizeof(struct fann_neuron));

    for(layer_it = ann->first_layer; layer_it != ann->last_layer; layer_it++){
        num_neurons = layer_it->last_neuron - layer_it->first_neuron;
        layer_it->first_neuron = neurons+num_neurons_so_far;
        layer_it->last_neuron = layer_it->first_neuron+num_neurons;
        num_neurons_so_far += num_neurons;
    }
```

```
    ann→output = (fann_type *)calloc(num_neurons, sizeof(fann_type));
}

/* Allocate room for the connections.
 */
void fann_allocate_connections(struct fann *ann)
{
    struct fann_layer *layer_it, *last_layer;
    struct fann_neuron *neuron_it, *last_neuron;
    fann_type *weights;
    struct fann_neuron **connected_neurons = NULL;
    unsigned int connections_so_far = 0;

    weights = (fann_type *)calloc(ann→total_connections, sizeof(fann_type));

    /* TODO make special cases for all places where the connections
       is used, so that it is not needed for fully connected networks.
    */
    connected_neurons = (struct fann_neuron **) calloc(ann→total_connections, sizeof(struct
fann_neuron*));

    last_layer = ann→last_layer;
    for(layer_it = ann→first_layer+1; layer_it ≠ ann→last_layer; layer_it++){
        last_neuron = layer_it→last_neuron-1;
        for(neuron_it = layer_it→first_neuron; neuron_it ≠ last_neuron; neuron_it++){
            neuron_it→weights = weights+connections_so_far;
            neuron_it→connected_neurons = connected_neurons+connections_so_far;
            connections_so_far += neuron_it→num_connections;
        }
    }

    if(connections_so_far ≠ ann→total_connections){
        printf("ERROR connections_so_far=%d, total_connections=%d\n", connections_so_far,
ann→total_connections);
        exit(0);
    }
}

/* Used to save the network to a file.
 */
int fann_save_internal(struct fann *ann, const char *configuration_file, unsigned int save_as_fixed)
{
    struct fann_layer *layer_it;
    int calculated_decimal_point = 0;
    struct fann_neuron *neuron_it, *first_neuron;
    fann_type *weights;
    struct fann_neuron **connected_neurons;
    unsigned int i = 0;
#ifndef FIXEDFANN
    /* variabels for use when saving floats as fixed point variabels */
    unsigned int decimal_point = 0;
    unsigned int fixed_multiplier = 0;
    fann_type max_possible_value = 0;
    unsigned int bits_used_for_max = 0;
    fann_type current_max_value = 0;
#endif

    FILE *conf = fopen(configuration_file, "w+");
    if(!conf){
        printf("Unable to open configuration file \"%s\" for writing.\n", configuration_file);
        return -1;
    }

#ifndef FIXEDFANN
    if(save_as_fixed){
        /* save the version information */
        fprintf(conf, FANN_FIX_VERSION"\n");
    }else{
        /* save the version information */
        fprintf(conf, FANN_FLO_VERSION"\n");
    }
#else
    /* save the version information */
    fprintf(conf, FANN_FIX_VERSION"\n");
#endif

#ifndef FIXEDFANN
    if(save_as_fixed){
        /* calculate the maximal possible shift value */
```

```
        for(layer_it = ann->first_layer+1; layer_it != ann->last_layer; layer_it++){
            for(neuron_it = layer_it->first_neuron; neuron_it != layer_it->last_neuron; neuron_it++){
                /* look at all connections to each neurons, and see how high a value we can get */
                current_max_value = 0;
                for(i = 0; i != neuron_it->num_connections; i++){
                    current_max_value += fann_abs(neuron_it->weights[i]);
                }

                if(current_max_value > max_possible_value){
                    max_possible_value = current_max_value;
                }
            }
        }

        for(bits_used_for_max = 0; max_possible_value >= 1; bits_used_for_max++){
            max_possible_value /= 2.0;
        }

        /* The maximum number of bits we shift the fix point, is the number
           of bits in a integer, minus one for the sign, one for the minus
           in stepwise sigmoid, and minus the bits used for the maximum.
           This is devided by two, to allow multiplication of two fixed
           point numbers.
        */
        calculated_decimal_point = (sizeof(int)*8-2-bits_used_for_max)/2;

        if(calculated_decimal_point < 0){
            decimal_point = 0;
        }else{
            decimal_point = calculated_decimal_point;
        }

        fixed_multiplier = 1 << decimal_point;

#ifdef DEBUG
        printf("calculated_decimal_point=%d, decimal_point=%u, bits_used_for_max=%u\n",
calculated_decimal_point, decimal_point, bits_used_for_max);
#endif

        /* save the decimal_point on a seperate line */
        fprintf(conf, "%u\n", decimal_point);

        /* save the number layers "num_layers learning_rate connection_rate activa-
tion_function_hidden activation_function_output activation_hidden_steepness activa-
tion_output_steepness" */

        fprintf(conf, "%u %f %f %u %u %d %d\n", ann->last_layer - ann->first_layer, ann->learning_rate,
ann->connection_rate, ann->activation_function_hidden, ann->activation_function_output,
(int)(ann->activation_hidden_steepness * fixed_multiplier), (int)(ann->activation_output_steepness *
fixed_multiplier));
    }else{
        /* save the number layers "num_layers learning_rate connection_rate activa-
tion_function_hidden activation_function_output activation_hidden_steepness activa-
tion_output_steepness" */

        fprintf(conf, "%u %f %f %u %u "FANNPRINTF" "FANNPRINTF"\n", ann->last_layer -
ann->first_layer, ann->learning_rate, ann->connection_rate, ann->activation_function_hidden,
ann->activation_function_output, ann->activation_hidden_steepness, ann->activation_output_steepness);
    }
#else
    /* save the decimal_point on a seperate line */
    fprintf(conf, "%u\n", ann->decimal_point);

    /* save the number layers "num_layers learning_rate connection_rate activa-
tion_function_hidden activation_function_output activation_hidden_steepness activa-
tion_output_steepness" */

    fprintf(conf, "%u %f %f %u %u "FANNPRINTF" "FANNPRINTF"\n", ann->last_layer -
ann->first_layer, ann->learning_rate, ann->connection_rate, ann->activation_function_hidden,
ann->activation_function_output, ann->activation_hidden_steepness,
ann->activation_output_steepness);
#endif

    for(layer_it = ann->first_layer; layer_it != ann->last_layer; layer_it++){
        /* the number of neurons in the layers (in the last layer, there is always one too many neurons, because of an unused
        fprintf(conf, "%u ", layer_it->last_neuron - layer_it->first_neuron);
    }
    fprintf(conf, "\n");
```

```
    for(layer_it = ann->first_layer; layer_it != ann->last_layer; layer_it++){
        /* the number of connections to each neuron */
        for(neuron_it = layer_it->first_neuron; neuron_it != layer_it->last_neuron; neuron_it++){
            fprintf(conf, "%u ", neuron_it->num_connections);
        }
        fprintf(conf, "\n");
    }

    connected_neurons = (ann->first_layer+1)->first_neuron->connected_neurons;
    weights = (ann->first_layer+1)->first_neuron->weights;
    first_neuron = ann->first_layer->first_neuron;

    /* Now save all the connections.
       We only need to save the source and the weight,
       since the destination is given by the order.

       The weight is not saved binary due to differences
       in binary definition of floating point numbers.
       Especially an iPAQ does not use the same binary
       representation as an i386 machine.
    */
    for(i = 0; i < ann->total_connections; i++){
#ifndef FIXEDFANN
        if(save_as_fixed){
            /* save the connection "(source weight) "*/
            fprintf(conf, "(%u %d) ",
                connected_neurons[i] - first_neuron,
                (int)floor((weights[i]*fixed_multiplier) + 0.5));
        }else{
            /* save the connection "(source weight) "*/
            fprintf(conf, "(%u "FANNPRINTF") ",
                connected_neurons[i] - first_neuron, weights[i]);
        }
#else
        /* save the connection "(source weight) "*/
        fprintf(conf, "(%u "FANNPRINTF") ",
            connected_neurons[i] - first_neuron, weights[i]);
#endif

    }
    fprintf(conf, "\n");

    fclose(conf);

    return calculated_decimal_point;
}

/* Save the train data structure.
 */
void fann_save_train_internal(struct fann_train_data* data, char *filename, unsigned int
save_as_fixed, unsigned int decimal_point)
{
    unsigned int num_data = data->num_data;
    unsigned int num_input = data->num_input;
    unsigned int num_output = data->num_output;
    unsigned int i, j;
#ifndef FIXEDFANN
    unsigned int multiplier = 1 << decimal_point;
#endif

    FILE *file = fopen(filename, "w");
    if(!file){
        printf("Unable to open train data file \"%s\" for writing.\n", filename);
        return;
    }

    fprintf(file, "%u %u %u\n", data->num_data, data->num_input, data->num_output);

    for(i = 0; i < num_data; i++){
        for(j = 0; j < num_input; j++){
#ifndef FIXEDFANN
            if(save_as_fixed){
                fprintf(file, "%d ", (int)(data->input[i][j]*multiplier));
            }else{
                fprintf(file, FANNPRINTF" ", data->input[i][j]);
            }
#else
            fprintf(file, FANNPRINTF" ", data->input[i][j]);
#endif
        }
```

```
        fprintf(file, "\n");

        for(j = 0; j < num_output; j++){
#ifndef FIXEDFANN
            if(save_as_fixed){
                fprintf(file, "%d ", (int)(data->output[i][j]*multiplier));
            }else{
                fprintf(file, FANNPRINTF" ", data->output[i][j]);
            }
#else
            fprintf(file, FANNPRINTF" ", data->output[i][j]);
#endif
        }
        fprintf(file, "\n");
    }

    fclose(file);
}

/* Seed the random function.
 */
void fann_seed_rand()
{
    FILE *fp = fopen("/dev/urandom", "r");
    unsigned int foo;
    struct timeval t;
    if(!fp){
        gettimeofday(&t, NULL);
        foo = t.tv_usec;
#ifdef DEBUG
        printf("unable to open /dev/urandom\n");
#endif
    }else{
        fread(&foo, sizeof(foo), 1, fp);
        fclose(fp);
    }
    srand(foo);
}
```

B.2 Test programs

B.2.1 xor_train.c

```
#include <stdio.h>

/* In this file I do not need to include floatfann or doublefann,
   because it is included in the makefile. Normaly you would need
   to do a #include "floatfann.h".
*/

int main()
{
    fann_type *calc_out;
    const float connection_rate = 1;
    const float learning_rate = 0.7;
    const unsigned int num_input = 2;
    const unsigned int num_output = 1;
    const unsigned int num_layers = 3;
    const unsigned int num_neurons_hidden = 4;
    const float desired_error = 0.0001;
    const unsigned int max_iterations = 500000;
    const unsigned int iterations_between_reports = 1000;
    struct fann *ann;
    struct fann_train_data *data;

    unsigned int i = 0;
    unsigned int decimal_point;

    printf("Creating network.\n");

    ann = fann_create(connection_rate, learning_rate, num_layers,
        num_input,
        num_neurons_hidden,
        num_output);

    printf("Training network.\n");

    data = fann_read_train_from_file("xor.data");
    fann_train_on_data(ann, data, max_iterations, iterations_between_reports, desired_error);

    printf("Testing network.\n");

    for(i = 0; i < data->num_data; i++){
        calc_out = fann_run(ann, data->input[i]);
        printf("XOR test (%f,%f) -> %f, should be %f, difference=%f\n",
        data->input[i][0], data->input[i][1], *calc_out, data->output[i][0], fann_abs(*calc_out -
data->output[i][0]));
    }

    printf("Saving network.\n");

    fann_save(ann, "xor_float.net");

    decimal_point = fann_save_to_fixed(ann, "xor_fixed.net");
    fann_save_train_to_fixed(data, "xor_fixed.data", decimal_point);

    printf("Cleaning up.\n");
    fann_destroy_train(data);
    fann_destroy(ann);

    return 0;
}
```

B.2.2 xor_test.c

```
#include <time.h>
#include <sys/time.h>
#include <stdio.h>

/* In this file I do not need to include fixedfann, floatfann or doublefann,
   because it is included in the makefile. Normaly you would need
   to do a #include "floatfann.h" or #include "fixedfann.h".
*/

int main()
{
    fann_type *calc_out;
    unsigned int i;
    int ret = 0;

    struct fann *ann;
    struct fann_train_data *data;

    printf("Creating network.\n");

#ifdef FIXEDFANN
    ann = fann_create_from_file("xor_fixed.net");
#else
    ann = fann_create_from_file("xor_float.net");
#endif

    if(!ann){
        printf("Error creating ann --- ABORTING.\n");
        return 0;
    }

    printf("Testing network.\n");

#ifdef FIXEDFANN
    data = fann_read_train_from_file("xor_fixed.data");
#else
    data = fann_read_train_from_file("xor.data");
#endif

    for(i = 0; i < data->num_data; i++){
        fann_reset_error(ann);
        calc_out = fann_test(ann, data->input[i], data->output[i]);
#ifdef FIXEDFANN
        printf("XOR test (%d, %d) -> %d, should be %d, difference=%f\n",
        data->input[i][0], data->input[i][1], *calc_out, data->output[i][0], (float)fann_abs(*calc_out -
data->output[i][0])/fann_get_multiplier(ann));

        if((float)fann_abs(*calc_out - data->output[i][0])/fann_get_multiplier(ann) > 0.1){
            printf("Test failed\n");
            ret = -1;
        }
#else
        printf("XOR test (%f, %f) -> %f, should be %f, difference=%f\n",
        data->input[i][0], data->input[i][1], *calc_out, data->output[i][0], (float)fann_abs(*calc_out -
data->output[i][0]));
#endif
    }

    printf("Cleaning up.\n");
    fann_destroy(ann);

    return ret;
}
```

B.2.3 steepness_train.c

```
/* In this file I do not need to include floatfann or doublefann,
   because it is included in the makefile. Normaly you would need
   to do a #include "floatfann.h".
*/

#include <stdio.h>

void train_on_steepness_file(struct fann *ann, char *filename,
    unsigned int max_epochs, unsigned int epochs_between_reports,
    float desired_error, float steepness_start,
    float steepness_step, float steepness_end)
{
    float error;
    unsigned int i, j;

    struct fann_train_data *data = fann_read_train_from_file(filename);
    if(epochs_between_reports){
        printf("Max epochs %8d.  Desired error:  %.10f\n",
            max_epochs, desired_error);
    }

    fann_set_activation_hidden_steepness(ann, steepness_start);
    fann_set_activation_output_steepness(ann, steepness_start);
    for(i = 1; i ≤ max_epochs; i++){
        /* train */
        fann_reset_error(ann);

        for(j = 0; j ≠ data→num_data; j++){
            fann_train(ann, data→input[j], data→output[j]);
        }

        error = fann_get_error(ann);

        /* print current output */
        if(epochs_between_reports &&
            (i % epochs_between_reports == 0
                || i == max_epochs
                || i == 1
                || error < desired_error)){
            printf("Epochs %8d.  Current error:  %.10f\n", i, error);
        }

        if(error < desired_error){
            steepness_start += steepness_step;
            if(steepness_start ≤ steepness_end){
                printf("Steepness:  %f\n", steepness_start);
                fann_set_activation_hidden_steepness(ann, steepness_start);
                fann_set_activation_output_steepness(ann, steepness_start);
            }else{
                break;
            }
        }
    }
    fann_destroy_train(data);
}

int main()
{
    const float connection_rate = 1;
    const float learning_rate = 0.7;
    const unsigned int num_input = 2;
    const unsigned int num_output = 1;
    const unsigned int num_layers = 3;
    const unsigned int num_neurons_hidden = 4;
    const float desired_error = 0.0001;
    const unsigned int max_iterations = 500000;
    const unsigned int iterations_between_reports = 1000;
    unsigned int i;
    fann_type *calc_out;

    struct fann_train_data *data;

    struct fann *ann = fann_create(connection_rate,
        learning_rate, num_layers,
        num_input, num_neurons_hidden, num_output);

    data = fann_read_train_from_file("xor.data");
```

```
    train_on_steepness_file(ann, "xor.data", max_iterations,
        iterations_between_reports, desired_error, 0.5, 0.1, 20.0);

    fann_set_activation_function_hidden(ann, FANN_THRESHOLD);
    fann_set_activation_function_output(ann, FANN_THRESHOLD);

    for(i = 0; i ≠ data→num_data; i++){
        calc_out = fann_run(ann, data→input[i]);
        printf("XOR test (%f, %f) -> %f, should be %f, difference=%f\n",
        data→input[i][0], data→input[i][1], *calc_out, data→output[i][0],
            (float)fann_abs(*calc_out - data→output[i][0]));
    }


    fann_save(ann, "xor_float.net");

    fann_destroy(ann);
    fann_destroy_train(data);

    return 0;
}
```

B.3 Benchmark programs

B.3.1 quality.cc

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "nets/backprop.h"
#include "ctimer.h"
#include "lwneuralnet.h"

unsigned int num_errors = 0;
double error_value = 0;

void clear_error()
{
    num_errors = 0;
    error_value = 0;
}

void update_error(fann_type *output, fann_type *desired_output, unsigned int num_output)
{
    unsigned int i = 0;
    /* calculate the error */
    for(i = 0; i < num_output; i++){
        error_value += (desired_output[i] - output[i]) * (desired_output[i] - output[i]);
    }
    num_errors++;
}

double mean_error()
{
    return error_value/(double)num_errors;
}


void quality_benchmark_jneural(
    struct fann_train_data *train_data,
    struct fann_train_data *test_data,
    FILE *train_out, FILE *test_out,
    unsigned int num_input, unsigned int num_neurons_hidden1,
    unsigned int num_neurons_hidden2, unsigned int num_output,
    unsigned int seconds_of_training, unsigned int seconds_between_reports)
{
    float train_error, test_error;
    unsigned int i;
    unsigned int epochs = 0;
    double elapsed = 0;
    double total_elapsed = 0;
    fann_type *output;
    struct backprop *ann;

    if(num_neurons_hidden2){
        ann = new backprop(0.7, num_input, num_output, 2, num_neurons_hidden1,
num_neurons_hidden2);
    }else{
        ann = new backprop(0.7, num_input, num_output, 1, num_neurons_hidden1);
    }

    calibrate_timer();

    while(total_elapsed < (double)seconds_of_training){
        /* train */
        elapsed = 0;
        start_timer();
        while(elapsed < (double)seconds_between_reports){
            for(i = 0; i ≠ train_data→num_data; i++){
                ann→set_input(train_data→input[i]);
                ann→train_on(train_data→output[i]);
            }

            elapsed = time_elapsed();
            epochs++;
        }
        stop_timer();
        total_elapsed += getSecs();

        /* make report */
        clear_error();
```

```
        for(i = 0; i ≠ train_data→num_data; i++){
            ann→set_input(train_data→input[i]);
            output = ann→query_output();
            update_error(output, train_data→output[i], train_data→num_output);
        }
        train_error = mean_error();

        clear_error();
        for(i = 0; i ≠ test_data→num_data; i++){
            ann→set_input(test_data→input[i]);
            output = ann→query_output();
            update_error(output, test_data→output[i], test_data→num_output);
        }
        test_error = mean_error();

        fprintf(train_out, "%f %.20e %d\n", total_elapsed, train_error, epochs);
        fprintf(test_out, "%f %.20e %d\n", total_elapsed, test_error, epochs);
        fprintf(stderr, ".");
    }

    delete ann;
}

void quality_benchmark_fann(float connection_rate,
    char *filename,
    struct fann_train_data *train_data,
    struct fann_train_data *test_data,
    FILE *train_out, FILE *test_out,
    unsigned int num_input, unsigned int num_neurons_hidden1,
    unsigned int num_neurons_hidden2, unsigned int num_output,
    unsigned int seconds_of_training, unsigned int seconds_between_reports)
{
    float train_error, test_error;
    unsigned int i, decimal_point;
    unsigned int epochs = 0;
    double elapsed = 0;
    double total_elapsed = 0;
    fann_type *output;
    struct fann *ann;
    char fixed_point_file[256];

    if(num_neurons_hidden2){
        ann = fann_create(connection_rate, 0.7, 4,
            num_input, num_neurons_hidden1, num_neurons_hidden2, num_output);
    }else{
        ann = fann_create(connection_rate, 0.7, 3,
            num_input, num_neurons_hidden1, num_output);
    }

    calibrate_timer();

    while(total_elapsed < (double)seconds_of_training){
        /* train */
        elapsed = 0;
        start_timer();
        while(elapsed < (double)seconds_between_reports){
            for(i = 0; i ≠ train_data→num_data; i++){
                fann_train(ann, train_data→input[i], train_data→output[i]);
            }

            elapsed = time_elapsed();
            epochs++;
        }
        stop_timer();
        total_elapsed += getSecs();

        /* make report */
        clear_error();
        for(i = 0; i ≠ train_data→num_data; i++){
            output = fann_run(ann, train_data→input[i]);
            update_error(output, train_data→output[i], train_data→num_output);
        }
        train_error = mean_error();

        clear_error();
        for(i = 0; i ≠ test_data→num_data; i++){
            output = fann_run(ann, test_data→input[i]);
            update_error(output, test_data→output[i], test_data→num_output);
        }
        test_error = mean_error();
```

```
            fprintf(train_out, "%f %.20e %d\n", total_elapsed, train_error, epochs);
            fprintf(test_out, "%f %.20e %d\n", total_elapsed, test_error, epochs);
            fprintf(stderr, ".");

            /* Save the data as fixed point, to allow for drawing of
               a fixed point graph */
            if(connection_rate == 1){
                /* buffer overflow could occur here */
                sprintf(fixed_point_file, "%05d_%f_%s_fixed", epochs, total_elapsed, filename);
                decimal_point = fann_save_to_fixed(ann, fixed_point_file);

                sprintf(fixed_point_file, "%s_fixed_train_%d", filename, decimal_point);
                fann_save_train_to_fixed(train_data, fixed_point_file, decimal_point);

                sprintf(fixed_point_file, "%s_fixed_test_%d", filename, decimal_point);
                fann_save_train_to_fixed(test_data, fixed_point_file, decimal_point);
            }
        }

        fann_destroy(ann);
}

void quality_benchmark_lwnn(
        struct fann_train_data *train_data,
        struct fann_train_data *test_data,
        FILE *train_out, FILE *test_out,
        unsigned int num_input, unsigned int num_neurons_hidden1,
        unsigned int num_neurons_hidden2, unsigned int num_output,
        unsigned int seconds_of_training, unsigned int seconds_between_reports)
{
        float train_error = 0;
        float test_error = 0;
        unsigned int i;
        unsigned int epochs = 0;
        double elapsed = 0;
        double total_elapsed = 0;
        fann_type *output;
        network_t *ann;

        if(num_neurons_hidden2){
            ann = net_allocate (4, num_input, num_neurons_hidden1, num_neurons_hidden2, num_output);
        }else{
            ann = net_allocate (3, num_input, num_neurons_hidden1, num_output);
        }

        net_set_learning_rate(ann, 0.7);

        calibrate_timer();

        output = (fann_type *)calloc(num_output, sizeof(fann_type));

        while(total_elapsed < (double)seconds_of_training){
            /* train */
            elapsed = 0;
            start_timer();
            while(elapsed < (double)seconds_between_reports){
                for(i = 0; i ≠ train_data→num_data; i++){
                    /* compute the outputs for inputs(i) */
                    net_compute (ann, train_data→input[i], output);

                    /* find the error with respect to targets(i) */
                    net_compute_output_error (ann, train_data→output[i]);

                    /* train the network one step */
                    net_train (ann);
                }

                elapsed = time_elapsed();
                epochs++;
            }
            stop_timer();
            total_elapsed += getSecs();

            /* make report */

            clear_error();
            for(i = 0; i ≠ train_data→num_data; i++){
                net_compute (ann, train_data→input[i], output);
                update_error(output, train_data→output[i], train_data→num_output);
```

```
        }
        train_error = mean_error();

        clear_error();
        for(i = 0; i ≠ test_data→num_data; i++){
            net_compute (ann, test_data→input[i], output);
            update_error(output, test_data→output[i], test_data→num_output);
        }
        test_error = mean_error();


        fprintf(train_out, "%f %.20e %d\n", total_elapsed, train_error, epochs);
        fprintf(test_out, "%f %.20e %d\n", total_elapsed, test_error, epochs);
        fprintf(stderr, ".");
    }

    net_free(ann);
}

int main(int argc, char* argv[])
{
    /* parameters */
    unsigned int num_neurons_hidden1;
    unsigned int num_neurons_hidden2;
    unsigned int seconds_of_training;
    unsigned int seconds_between_reports;

    struct fann_train_data *train_data, *test_data;
    FILE *train_out, *test_out;

    if(argc ≠ 10){
        printf("usage %s net train_file test_file train_file_out test_file_out num_hidden1
num_hidden2 seconds_of_training seconds_between_reports\n", argv[0]);
        return -1;
    }

    num_neurons_hidden1 = atoi(argv[6]);
    num_neurons_hidden2 = atoi(argv[7]);
    seconds_of_training = atoi(argv[8]);
    seconds_between_reports = atoi(argv[9]);

    train_data = fann_read_train_from_file(argv[2]);
    test_data = fann_read_train_from_file(argv[3]);

    if(strlen(argv[4]) == 1 && argv[4][0] == '-'){
        train_out = stdout;
    }else{
        train_out = fopen(argv[4], "w");
    }

    if(strlen(argv[5]) == 1 && argv[5][0] == '-'){
        test_out = stdout;
    }else{
        test_out = fopen(argv[5], "w");
    }

    fprintf(stderr, "Quality test of %s %s ", argv[1], argv[2]);

    if(strcmp(argv[1], "lwnn") == 0){
        quality_benchmark_lwnn(train_data, test_data,
            train_out, test_out,
            train_data→num_input, num_neurons_hidden1,
            num_neurons_hidden2, train_data→num_output,
            seconds_of_training, seconds_between_reports);
    }else if(strcmp(argv[1], "fann") == 0){
        quality_benchmark_fann(1, argv[4], train_data, test_data,
            train_out, test_out,
            train_data→num_input, num_neurons_hidden1,
            num_neurons_hidden2, train_data→num_output,
            seconds_of_training, seconds_between_reports);
    }else if(strcmp(argv[1], "fann_half") == 0){
        quality_benchmark_fann(0.75, NULL, train_data, test_data,
            train_out, test_out,
            train_data→num_input, num_neurons_hidden1,
            num_neurons_hidden2, train_data→num_output,
            seconds_of_training, seconds_between_reports);
    }else if(strcmp(argv[1], "jneural") == 0){
        quality_benchmark_jneural(train_data, test_data,
            train_out, test_out,
            train_data→num_input, num_neurons_hidden1,
```

```
            num_neurons_hidden2, train_data->num_output,
            seconds_of_training, seconds_between_reports);
    }

    fprintf(stderr, "\n");

    fann_destroy_train(train_data);
    fann_destroy_train(test_data);

    return 0;
}
```

B.3.2 quality_fixed.c

```
#include <stdio.h>
#include "fixedfann.h"

int main(int argc, char* argv[])
{
    struct fann_train_data *train_data, *test_data;
    FILE *train_out, *test_out;
    struct fann *ann;
    float train_error, test_error;
    unsigned int i, j;
    unsigned int epochs = 0;
    double total_elapsed = 0;
    char file[256];

    if(argc < 6){
        printf("usage %s train_file test_file train_file_out test_file_out fixed_conf_files\n",
argv[0]);
        return -1;
    }

    if(strlen(argv[3]) == 1 && argv[3][0] == '-'){
        train_out = stdout;
    }else{
        train_out = fopen(argv[3], "w");
    }

    if(strlen(argv[4]) == 1 && argv[4][0] == '-'){
        test_out = stdout;
    }else{
        test_out = fopen(argv[4], "w");
    }

    for(j = 5; j < argc; j++){
        ann = fann_create_from_file(argv[j]);

        sprintf(file, "%s_%d", argv[1], fann_get_decimal_point(ann));
        train_data = fann_read_train_from_file(file);

        sprintf(file, "%s_%d", argv[2], fann_get_decimal_point(ann));
        test_data = fann_read_train_from_file(file);

        fann_reset_error(ann);
        for(i = 0; i ≠ train_data→num_data; i++){
            fann_test(ann, train_data→input[i], train_data→output[i]);
        }
        train_error = fann_get_error(ann);

        fann_reset_error(ann);
        for(i = 0; i ≠ test_data→num_data; i++){
            fann_test(ann, test_data→input[i], test_data→output[i]);
        }
        test_error = fann_get_error(ann);

        sscanf(argv[j], "%d_%lf", &epochs, &total_elapsed);
        fprintf(train_out, "%f %.20e %d\n", total_elapsed, train_error, epochs);
        fprintf(test_out, "%f %.20e %d\n", total_elapsed, test_error, epochs);
        fprintf(stderr, ".");

        fann_destroy(ann);
    }

    return 0;
}
```

B.3.3 performance.cc

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "ctimer.h"

#ifndef FIXEDFANN

#include "nets/backprop.h"
#include "lwneuralnet.h"

void performance_benchmark_jneural(FILE *out, fann_type *input,
    unsigned int num_neurons, unsigned int seconds_per_test)
{
    unsigned int i, total_connections;
    fann_type *output;

    backprop *ann = new backprop(0.7, num_neurons, num_neurons, 2, num_neurons, num_neurons);

    total_connections = (num_neurons+1) * num_neurons * 3;

    start_timer();

    for(i = 0; time_elapsed() < (double)seconds_per_test; i++){
        ann->set_input(input);
        output = ann->query_output();
    }

    stop_timer();

    fprintf(out, "%d %.20e\n", num_neurons, getNanoPerN(i)/total_connections);
    fprintf(stderr, "%d ", num_neurons);

    delete ann;
}

void performance_benchmark_lwnn(FILE *out, fann_type *input,
    unsigned int num_neurons, unsigned int seconds_per_test)
{
    unsigned int i, total_connections;
    fann_type *output;

    output = (fann_type*)calloc(num_neurons, sizeof(fann_type));
    network_t *ann = net_allocate (4, num_neurons, num_neurons, num_neurons, num_neurons);

    total_connections = num_neurons * num_neurons * 3;

    start_timer();

    for(i = 0; time_elapsed() < (double)seconds_per_test; i++){
        net_compute (ann, input, output);
    }

    stop_timer();

    fprintf(out, "%d %.20e\n", num_neurons, getNanoPerN(i)/total_connections);
    fprintf(stderr, "%d ", num_neurons);

    net_free(ann);
    free(output);
}

void performance_benchmark_fann_noopt(FILE *out, fann_type *input,
    unsigned int num_neurons, unsigned int seconds_per_test)
{
    unsigned int i, total_connections;
    fann_type *output;

    struct fann *ann = fann_create(1, 0.7, 4,
        num_neurons, num_neurons, num_neurons, num_neurons);

    //just to fool the optimizer into thinking that the network is not fully connected
    ann->connection_rate = 0.9;

    total_connections = (num_neurons+1) * num_neurons * 3;

    start_timer();

    for(i = 0; time_elapsed() < (double)seconds_per_test; i++){
```

```
        output = fann_run(ann, input);
    }

    stop_timer();

    fprintf(out, "%d %.20e\n", num_neurons, getNanoPerN(i)/total_connections);
    fprintf(stderr, "%d ", num_neurons);
    fann_destroy(ann);
}

void performance_benchmark_fann_thres(FILE *out, fann_type *input,
    unsigned int num_neurons, unsigned int seconds_per_test)
{
    unsigned int i, total_connections;
    fann_type *output;

    struct fann *ann = fann_create(1, 0.7, 4,
        num_neurons, num_neurons, num_neurons, num_neurons);

    fann_set_activation_function_hidden(ann, FANN_THRESHOLD);
    fann_set_activation_function_output(ann, FANN_THRESHOLD);

    total_connections = (num_neurons+1) * num_neurons * 3;

    start_timer();

    for(i = 0; time_elapsed() < (double)seconds_per_test; i++){
        output = fann_run(ann, input);
    }

    stop_timer();

    fprintf(out, "%d %.20e\n", num_neurons, getNanoPerN(i)/total_connections);
    fprintf(stderr, "%d ", num_neurons);
    fann_destroy(ann);
}

#endif

void performance_benchmark_fann(FILE *out, fann_type *input,
    unsigned int num_neurons, unsigned int seconds_per_test)
{
    unsigned int i, total_connections;
    fann_type *output;

    struct fann *ann = fann_create(1, 0.7, 4,
        num_neurons, num_neurons, num_neurons, num_neurons);

    total_connections = (num_neurons+1) * num_neurons * 3;

    start_timer();

    for(i = 0; time_elapsed() < (double)seconds_per_test; i++){
        output = fann_run(ann, input);
    }

    stop_timer();

    fprintf(out, "%d %.20e\n", num_neurons, getNanoPerN(i)/total_connections);
    fprintf(stderr, "%d ", num_neurons);
    fann_destroy(ann);
}

int main(int argc, char* argv[])
{
    /* parameters */
    unsigned int num_neurons_first;
    unsigned int num_neurons_last;
    double multiplier;
    unsigned int seconds_per_test;
    FILE *out;

    fann_type *input;
    unsigned int num_neurons, i;

    if(argc ≠ 7){
        printf("usage %s net file_out num_neurons_first num_neurons_last multiplier
seconds_per_test\n", argv[0]);
        return -1;
    }
```

```
    calibrate_timer();

    num_neurons_first = atoi(argv[3]);
    num_neurons_last = atoi(argv[4]);
    multiplier = atof(argv[5]);
    seconds_per_test = atoi(argv[6]);

    if(strlen(argv[2]) == 1 && argv[2][0] == '-'){
        out = stdout;
    }else{
        out = fopen(argv[2], "w");
    }

    fprintf(stderr, "Performance test of %s %s ", argv[1], argv[2]);

    input = (fann_type*)calloc(num_neurons_last, sizeof(fann_type));
    for(i = 0; i < num_neurons_last; i++){
        input[i] = fann_random_weight();                    //fill input with random variables
    }

    for(num_neurons = num_neurons_first;
        num_neurons ≤ num_neurons_last; num_neurons = (int)(num_neurons * multiplier)){

#ifndef FIXEDFANN
        if(strcmp(argv[1], "lwnn") == 0){
            performance_benchmark_lwnn(out, input,
              num_neurons, seconds_per_test);
        }else if(strcmp(argv[1], "fann") == 0){
#endif
            performance_benchmark_fann(out, input,
                num_neurons, seconds_per_test);
#ifndef FIXEDFANN
        }else if(strcmp(argv[1], "fann_noopt") == 0){
            performance_benchmark_fann_noopt(out, input,
                num_neurons, seconds_per_test);
        }else if(strcmp(argv[1], "fann_thres") == 0){
            performance_benchmark_fann_thres(out, input,
                num_neurons, seconds_per_test);
        }else if(strcmp(argv[1], "jneural") == 0){
            performance_benchmark_jneural(out, input,
                num_neurons, seconds_per_test);
        }
#endif

    }

    fprintf(stderr, "\n");
    free(input);

    return 0;
}
```

B.3.4 benchmark.sh

```
#!/bin/sh
test/performance fann fann_performance.out 1 2048 2 20
test/performance fann_noopt fann_noopt_performance.out 1 2048 2 20
test/performance fann_thres fann_thres_performance.out 1 2048 2 20
test/performance_fixed fann fann_fixed_performance.out 1 2048 2 20
test/performance lwnn lwnn_performance.out 1 2048 2 20
test/performance jneural jneural_performance.out 1 512 2 20

#./performance_arm fann fann_performance_arm.out 1 512 2 20
#./performance_arm fann_noopt fann_noopt_performance_arm.out 1 512 2 20
#./performance_arm fann_thres fann_thres_performance_arm.out 1 512 2 20
#./performance_fixed_arm fann fann_fixed_performance_arm.out 1 512 2 20
#./performance_arm lwnn lwnn_performance_arm.out 1 512 2 20
#./performance_arm jneural jneural_performance_arm.out 1 512 2 20

rm -f *_fixed
test/quality fann datasets/mydata/building.train datasets/mydata/building.test
building_fann_train.out building_fann_test.out 16 0 200 1
test/quality_fixed building_fann_train.out_fixed_train building_fann_train.out_fixed_test
building_fann_fixed_train.out building_fann_fixed_test.out *_fixed
test/quality fann_half datasets/mydata/building.train datasets/mydata/building.test
building_fann_half_train.out building_fann_half_test.out 16 0 200 1
test/quality lwnn datasets/mydata/building.train datasets/mydata/building.test
building_lwnn_train.out building_lwnn_test.out 16 0 200 1
test/quality jneural datasets/mydata/building.train datasets/mydata/building.test
building_jneural_train.out building_jneural_test.out 16 0 200 1

rm -f *_fixed
test/quality fann datasets/mydata/card.train datasets/mydata/card.test card_fann_train.out
card_fann_test.out 32 0 200 1
test/quality_fixed card_fann_train.out_fixed_train card_fann_train.out_fixed_test
card_fann_fixed_train.out card_fann_fixed_test.out *_fixed
test/quality fann_half datasets/mydata/card.train datasets/mydata/card.test card_fann_half_train.out
card_fann_half_test.out 32 0 200 1
test/quality lwnn datasets/mydata/card.train datasets/mydata/card.test card_lwnn_train.out
card_lwnn_test.out 32 0 200 1
test/quality jneural datasets/mydata/card.train datasets/mydata/card.test card_jneural_train.out
card_jneural_test.out 32 0 200 1

rm -f *_fixed
test/quality fann datasets/mydata/gene.train datasets/mydata/gene.test gene_fann_train.out
gene_fann_test.out 4 2 200 1
test/quality_fixed gene_fann_train.out_fixed_train gene_fann_train.out_fixed_test
gene_fann_fixed_train.out gene_fann_fixed_test.out *_fixed
test/quality fann_half datasets/mydata/gene.train datasets/mydata/gene.test gene_fann_half_train.out
gene_fann_half_test.out 4 2 200 1
test/quality lwnn datasets/mydata/gene.train datasets/mydata/gene.test gene_lwnn_train.out
gene_lwnn_test.out 4 2 200 1
test/quality jneural datasets/mydata/gene.train datasets/mydata/gene.test gene_jneural_train.out
gene_jneural_test.out 4 2 200 1

rm -f *_fixed
test/quality fann datasets/mydata/mushroom.train datasets/mydata/mushroom.test
mushroom_fann_train.out mushroom_fann_test.out 32 0 200 1
test/quality_fixed mushroom_fann_train.out_fixed_train mushroom_fann_train.out_fixed_test
mushroom_fann_fixed_train.out mushroom_fann_fixed_test.out *_fixed
test/quality fann_half datasets/mydata/mushroom.train datasets/mydata/mushroom.test
mushroom_fann_half_train.out mushroom_fann_half_test.out 32 0 200 1
test/quality lwnn datasets/mydata/mushroom.train datasets/mydata/mushroom.test
mushroom_lwnn_train.out mushroom_lwnn_test.out 32 0 200 1
test/quality jneural datasets/mydata/mushroom.train datasets/mydata/mushroom.test
mushroom_jneural_train.out mushroom_jneural_test.out 32 0 200 1

rm -f *_fixed
test/quality fann datasets/mydata/soybean.train datasets/mydata/soybean.test
soybean_fann_train.out soybean_fann_test.out 16 8 200 1
test/quality_fixed soybean_fann_train.out_fixed_train soybean_fann_train.out_fixed_test
soybean_fann_fixed_train.out soybean_fann_fixed_test.out *_fixed
test/quality fann_half datasets/mydata/soybean.train datasets/mydata/soybean.test
soybean_fann_half_train.out soybean_fann_half_test.out 16 8 200 1
test/quality lwnn datasets/mydata/soybean.train datasets/mydata/soybean.test
soybean_lwnn_train.out soybean_lwnn_test.out 16 8 200 1
test/quality jneural datasets/mydata/soybean.train datasets/mydata/soybean.test
soybean_jneural_train.out soybean_jneural_test.out 16 8 200 1

rm -f *_fixed
test/quality fann datasets/mydata/thyroid.train datasets/mydata/thyroid.test thyroid_fann_train.out
thyroid_fann_test.out 16 8 200 1
```

```
test/quality_fixed thyroid_fann_train.out_fixed_train thyroid_fann_train.out_fixed_test
thyroid_fann_fixed_train.out thyroid_fann_fixed_test.out *_fixed
test/quality fann_half datasets/mydata/thyroid.train datasets/mydata/thyroid.test
thyroid_fann_half_train.out thyroid_fann_half_test.out 16 8 200 1
test/quality lwnn datasets/mydata/thyroid.train datasets/mydata/thyroid.test thyroid_lwnn_train.out
thyroid_lwnn_test.out 16 8 200 1
test/quality jneural datasets/mydata/thyroid.train datasets/mydata/thyroid.test
thyroid_jneural_train.out thyroid_jneural_test.out 16 8 200 1
```