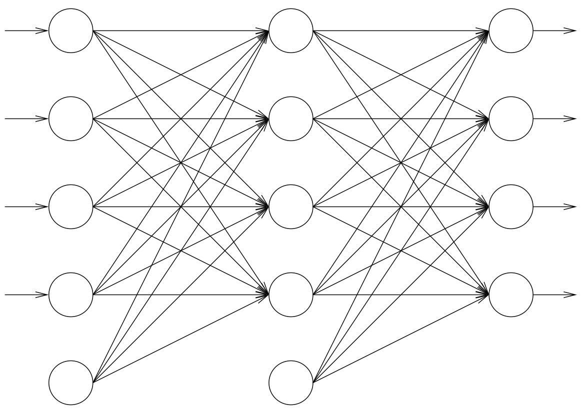
# Implementation of a Fast Artificial Neural Network Library (fann)

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.

# 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.