Neural Networks Made Simple

Steffen Nissen

On the CD:

Full documentation concerning the Fann library (the files including source code as well as the Fann library version) can be found on the CD.

For years, the Hollywood science fiction films such as *I, Robot* have portrayed an artificial intelligence (AI) as a harbinger of Armageddon. Nothing, however, could be farther from the truth. While Hollywood regales us with horror stories of our imminent demise, people with no interest in the extinction of our species have been harnessing AI to make our lives easier, more productive, longer and generally better.

The robots in the *I, Robot* film have an artificial brain based on a network of artificial neurons; this artificial neural network (ANN) is built to model the human brain's own neural network. The Fast Artificial Neural Network (FANN) library is an ANN library, which can be used from C, C++, PHP, Python, Delphi and Mathematica and although, it cannot create Hollywood magic, it is still a powerful tool for software developers. ANNs can be used in areas as diverse as creating more appealing game-play in computer games, identifying objects in images and helping the stock brokers predict trends of the ever-changing stock market.

Function approximation

ANNs apply the principle of function approximation by example, meaning that they learn a function by looking at examples of this function. One of the simplest examples is an ANN learning the XOR function, but it could just as easily be learning to determine the language of a text, or whether there is a tumour visible in an X-ray image.

If an ANN is to be able to learn a problem, it must be defined as a function with a set of input and output variables supported by examples of how this function should work. A problem like the XOR function is already defined as a function with two binary input variables and a binary output variable, and with the examples which are defined by the results of four different input patterns. However, there are more complicated problems which can be more difficult to define as functions. The input variables to the problem of finding a tumour in an X-ray image could be the pixel values of the image, but they could also be some values extracted from the image. The output could then either be a binary value or a floating-point value representing the probability of a tumour in the image. In ANNs this floating-point value would normally be between 0 and 1, inclusive.

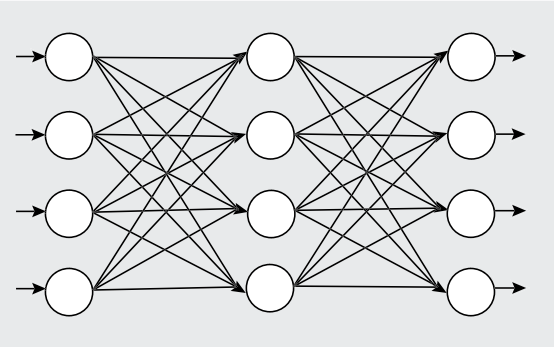

Figure 1. *An ANN with four input neurons, a hidden layer and four output neurons*

The human brain

A function approximator like an ANN can be viewed as a black box and when it comes to FANN, this is more or less all you will need to know. However, basic knowledge of how the human brain operates is needed to understand how ANNs work.

The human brain is a highly complicated system which is capable to solve very complex problems. The brain consists of many different elements, but one of its most important building blocks is the neuron, of which it contains approximately 10^{11}. These neurons are connected by around 10^{15} connections, creating a huge neural network. Neurons send impulses to each other through the connections and these impulses make the brain work. The neural network also receives impulses from the five senses and sends out impulses to muscles to achieve motion or speech.

The individual neuron can be seen as an input-output machine which waits for impulses from the surrounding neurons and, when it has received enough impulses, it sends out an impulse to other neurons.

Artificial Neural Networks

Artificial neurons are similar to their biological counterparts. They have input connections which are summed together to determine the strength of their output, which is the result of the sum being fed into an activation function. Though many activation functions exist, the most common is the sigmoid activation function, which outputs a number between 0 (for low input values) and 1 (for high input values). The resultant of this function is then passed as the input to other neurons through more connections, each of which are weighted. These weights determine the behaviour of the network.

Steffen Nissen is a Computer Scientist from Denmark. He has created and actively maintained the FANN library, while the others maintain the bindings for other languages. He is also the author of technical report covering the creation of FANN library, *Implementation of a Fast Artificial Neural Network Library (fann)*.
Contact the author: *lukesky@diku.dk*

Listing 1. *Program that calculates the frequencies of the letters A-Z in a text file*

```
#include <vector>
#include <fstream>
#include <iostream>
#include <ctype.h>

void error(const char* p, const char* p2 = "")
{
   std::cerr << p << ' ' << p2 << std::endl;
   std::exit(1);
}

void generate_frequencies(const char *filename,
   float *frequencies)
{
   std::ifstream infile(filename);
   if(!infile) error("Cannot open input file", filename);

   std::vector<unsigned int> letter_count(26, 0);
   unsigned int num_characters = 0;
   char c;
   while(infile.get(c)){
      c = tolower(c);
      if(c >= 'a' && c <= 'z'){
         letter_count[c - 'a']++;
         num_characters++;
      }
   }

   if(!infile.eof()) error("Something strange happened");
   for(unsigned int i = 0; i != 26; i++){
      frequencies[i] =
         letter_count[i]/(double)num_characters;
   }
}

int main(int argc, char* argv[])
{
   if(argc != 2) error("Remember to specify an input file");

   float frequencies[26];
   generate_frequencies(argv[1], frequencies);

   for(unsigned int i = 0; i != 26; i++){
      std::cout << frequencies[i] << ' ';
   }
   std::cout << std::endl;

   return 0;
}
```

In the human brain the neurons are connected in a seemingly random order and send impulses asynchronously. If we wanted to model a brain this might be the way to organise an ANN, but since we primarily want to create a function approximator, ANNs are usually not organised like this.

When we create ANNs, the neurons are usually ordered in layers with connections going between the layers. The first layer contains the input neurons and the last layer contains the output neurons. These input and output neurons represent the input and output variables of the function that we want to approximate. Between the input and the output layer a number of hidden layers exist and the connections (and weights) to and from these hidden layers determine how well the ANN performs. When an ANN is learning to approximate a function, it is shown examples of how the function works and the internal weights in the ANN are slowly adjusted so as to produce the same output as in the examples. The hope is that when the ANN is shown a new set of input variables, it will give a correct output. Therefore, if an ANN is expected to learn to spot a tumour in an X-ray image, it will be shown many X-ray images containing tumours, and many X-ray images containing healthy tissues. After a period of training with these images, the weights in the ANN should hopefully contain information which will allow it to positively identify tumours in X-ray images that it has not seen during the training.

A FANN library tutorial

The Internet has made global communication a part of many people's lives, but it has also given rise to the problem that everyone does not speak the same language. Translation tools can help bridge this gap, but in order for such tools to work they need to know in what language a passage of text is written. One way to determine this is by examining the frequency of letters occurring in a text. While this seems like a very naïve approach to language detection, it has proven to be very effective. For many European languages it is enough to look at the frequencies of the letters A to Z, even though some languages also use other letters as well. Easily enough, the

Listing 2. *The first part of the training file with character frequencies for English, French and Polish, the first line is a header telling that there are 12 training patterns consisting of 26 inputs and 3 outputs.*

```
12 26 3

0.103 0.016 0.054 0.060 0.113 0.010 0.010 0.048 0.056
0.003 0.010 0.035 0.014 0.065 0.075 0.013 0.000 0.051
0.083 0.111 0.030 0.008 0.019 0.000 0.016 0.000

1 0 0

0.076 0.010 0.022 0.039 0.151 0.013 0.009 0.009 0.081
0.001 0.000 0.058 0.024 0.074 0.061 0.030 0.011 0.069
0.100 0.074 0.059 0.015 0.000 0.009 0.003 0.003

0 1 0

0.088 0.016 0.030 0.034 0.089 0.004 0.011 0.023 0.071
0.032 0.030 0.025 0.047 0.058 0.093 0.040 0.000 0.062
0.044 0.035 0.039 0.002 0.044 0.000 0.037 0.046

0 0 1

...
```

FANN library can be used to make a small program that determines the language of a text file. The ANN used should have an input neuron for each of the 26 letters, and an output neuron for each of the languages. But first, a small program must be made measuring the frequency of the letters in a text file.

Listing 1 will generate letter frequencies for a file and output them in a format that can be used to generate a training file for the FANN library. Training files for the FANN library must consist of a line containing the input values, followed by a line containing the output values. If we wish to distinguish between three different languages (English, French and Polish), we could choose to represent this by allocating one output variable with a value of 0 for English, 0.5 for French and 1 for Polish. Neural networks are, however, known to perform better if an output variable is allocated for each language, and that it is set to 1 for the correct language and 0 otherwise.

With this small program at hand, a training file containing letter frequencies can be generated for texts written in the different languages. The ANN will of course be better at distinguishing the languages if frequencies for many different texts are available in the training file, but for this small example, 3-4 texts in each language should be enough. Listing 2 shows a pre-generated training file using 4 text files for each of the three languages and Figure 2 shows a graphical representation of the frequencies in the file. A thorough inspection of this file shows clear trends: English has more H's than the other two languages, French has almost no K's and Polish has more W's and Z's than the other languages. The training file only uses letters in the A to Z range, but since a language like Polish use letters like Ł, Ą and Ę which are not used in the other two languages, a more precise ANN could be made by adding input neurons for these letters as well. When only comparing three languages, there is, however, no need for these added letters since the remaining letters contain enough information to classify the languages correctly, but if the ANN were to classify hundreds of different languages, more letters would be required.

With a training file like this it is very easy to create a program using FANN which can be used to train an ANN to distinguish between the three languages. Listing 2 shows just how simply this can be done with FANN. This program uses four FANN functions `fann_create`, `fann_train_on_file`, `fann_save` and `fann_destroy`. The function `struct fann* fann_create(float connection_rate, float learning_rate, unsigned int num_layers, ...)` is used to create an ANN, where the `connection_rate` parameter can be used to create an ANN that is not fully connected, although fully connected ANNs are normally preferred, and the `learning_rate` is used to specify how aggressive the learning algorithm should be (only relevant for some learning algorithms). The last parameters for the function are used to define the layout of the layers in the ANN. In this case, an ANN with three layers (one input, one hidden and one output) has been chosen. The input layer has 26 neurons (one for each letter), the output layer has three neurons (one for each language) and the hidden layer has 13 neurons. The number of layers and number of neurons in the hidden layer has been selected experimentally, as there is really no easy way of determining these values. It helps, however, to remember that the ANN learns by adjusting the weights, so if an ANN contains more neurons and thereby also more weights it can learn

Listing 3. *A program that trains an ANN to learn to distinguish between languages*

```
#include "fann.h"

int main()
{
    struct fann *ann = fann_create(1, 0.7, 3, 26, 13, 3);
    fann_train_on_file(ann, "frequencies.data", 200, 10,
        0.0001);
    fann_save(ann, "language_classify.net");
    fann_destroy(ann);
    return 0;
}
```

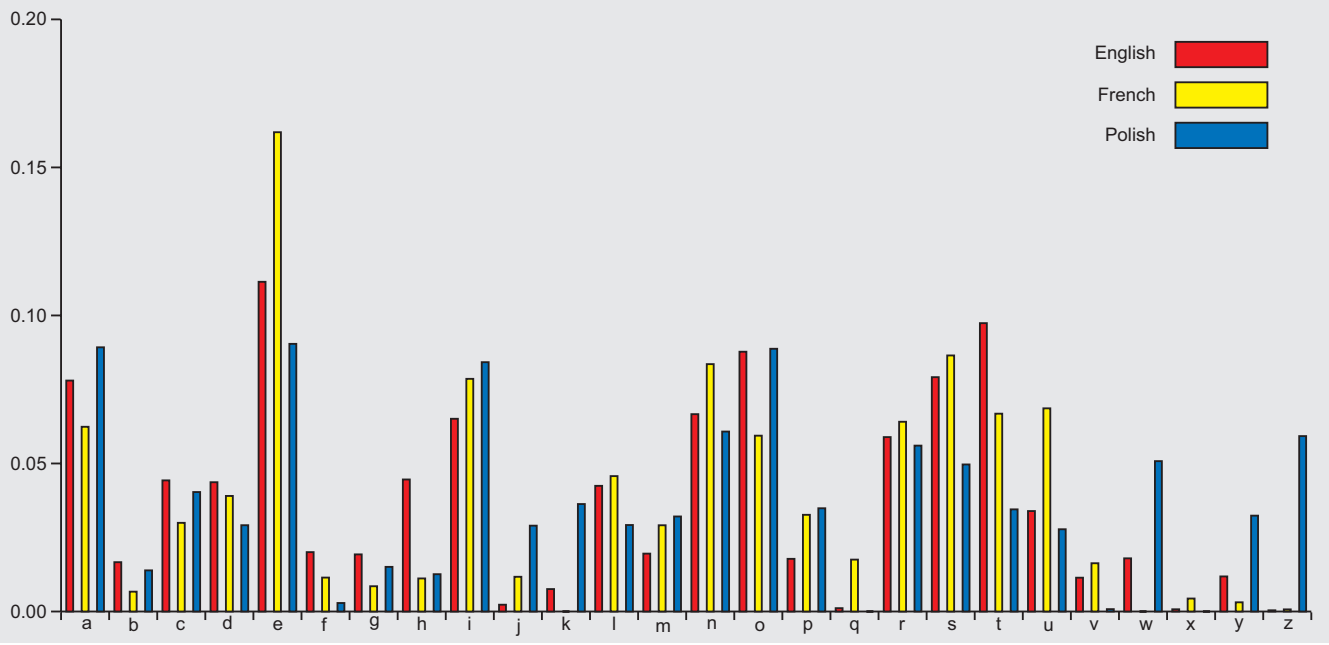

Figure 2. *A chart of the average frequencies of the letters in English, French and Polish*

Listing 4. *Output from FANN during training*

```
Max epochs      200. Desired error: 0.0001000000
Epochs            1. Current error: 0.7464869022
Epochs           10. Current error: 0.7226278782
Epochs           20. Current error: 0.6682052612
Epochs           30. Current error: 0.6573708057
Epochs           40. Current error: 0.5314316154
Epochs           50. Current error: 0.0589125119
Epochs           57. Current error: 0.0000702030
```

more complicated problems. Having too many weights can also be a problem, since learning can be more difficult and there is also a chance that the ANN will learn specific features of the input variables instead of general patterns which can be extrapolated to other data sets. In order for an ANN to accurately classify data not in the training set, this ability to generalise is crucial – without it, the ANN will be unable to distinguish frequencies that it has not been trained with.

The `void fann_train_on_file(struct fann *ann, char *filename, unsigned int max_epochs, unsigned int epochs_between_reports, float desired_error)` function trains the ANN. The training is done by continually adjusting the weights so that the output of the ANN matches the output in the training file. One cycle where the weights are adjusted to match the output in the training file is called an epoch. In this example the maximum number of epochs have been set to 200, and a status report is printed every 10 epochs. When measuring how close an ANN matches the desired output, the mean square error is usually used. The mean square error is the mean value of the squared difference between the actual and the desired output of the ANN, for individual training patterns. A small mean square error means a close match of the desired output.

When the program in Listing 2 is run, the ANN will be trained and some status information (see Listing 4) will be printed to make it easier to monitor progress during training. After training, the ANN could be used directly to determine which language a text is written in, but it is usually desirable to keep training and execution in two different programs, so that the more time-consuming training needs only be done only once. For this reason, Listing 2 simply saves the ANN to a file that can be loaded by another program.

The small program in Listing 5 loads the saved ANN and uses it to classify a text as English, French or Polish. When tested with texts in the three languages found on the Internet, it can properly classify texts as short as only a few sentences. Although this method for distinguishing between languages is not bullet-proof, I was not able to find a single text that could be classified incorrectly.

The FANN library: details

The language classification example shows just how easily the FANN library can be applied to solve simple, everyday computer science problems which would be much more difficult to solve using other methods. Unfortunately, not all problems can be solved this easily, and when working with ANNs one often finds oneself in a situation in which it is very difficult to train the ANN to give the correct results. Sometimes this is because the problem simply cannot be solved by ANNs, but often the training can be helped by tweaking the FANN library settings.

The most important factor when training an ANN is the size of the ANN. This can only be set experimentally, but knowledge of the problem will often help giving good guesses. With a reasonably sized ANN, the training can be done in many different ways. The FANN library supports several different training algorithms and the default algorithm (`FANN_TRAIN_RPROP`) might not always be the best-suited for a specific problem. If this is the case, the `fann_set_training_algorithm` function can be used to change the training algorithm. In version 1.2.0 of the FANN library there are four different training algorithms available, all of which use some sort of back propagation. Back-propagation algorithms change the weights by propagating the error backwards from the output layer to the input layer while adjusting the weights. The back-propagated error value could either be an error calculated for a single training pattern (incremental), or it could be a sum of errors from the entire training file (batch). `FANN_TRAIN_INCREMENTAL` implements an incremental training algorithm which alters the weights after each training pattern. The advantage of such a training algorithm is that the weights are being altered many times during each epoch and since each training pattern alters the weights in slightly different directions, the training will not easily get stuck in local minima – states in which all *small* changes in the weights only make the mean square error worse, even though the optimal solution may have not yet been found. `FANN_TRAIN_BATCH`, `FANN_TRAIN_RPROP` and `FANN_TRAIN_QUICKPROP` are all examples of batch-training algorithms which alter the weight after calculating the errors for an entire training set. The advantage of these algorithms is that they can make use of global optimisation information which is not available to incremental training algorithms. This can, however, mean that some of the finer points of the individual training patterns are being missed. There is no clear answer to the question which training algorithm is the best. One of the advanced batch-training algorithms like *rprop* or *quickprop* training is usually the best solution. Sometimes, however, incremental training is more optimal – especially if many

Listing 5. *A program classifying a text as written in one of the three languages (The program uses some functions defined in Listing 1)*

```
int main(int argc, char* argv[])
{
   if(argc != 2) error("Remember to specify an input file");
   struct fann *ann = fann_create_from_file(
      "language_classify.net");

   float frequencies[26];
   generate_frequencies(argv[1], frequencies);

   float *output = fann_run(ann, frequencies);
   std::cout << "English: " << output[0] << std::endl
             << "French : " << output[1] << std::endl
             << "Polish : " << output[2] << std::endl;

   return 0;
}
```

training patterns are available. In the language training example the most optimal training algorithm is the default *rprop* one, which reached the desired mean square error value after just 57 epochs. The incremental training algorithm needed 8108 epochs to reach the same result, while the batch training algorithm needed 91985 epochs. The *quickprop* training algorithm had more problems and at first it failed altogether at reaching the desired error value, but after tweaking the decay of the *quickprop* algorithm, it reached the desired error after 662 epochs. The decay of the *quickprop* algorithm is a parameter which is used to control how aggressive the *quickprop* training algorithm is and it can be altered by the `fann_set_quickprop_decay` function. Other `fann_set_...` functions can also be used to set additional parameters for the individual training algorithms, although some of these parameters can be a bit difficult to tweak without knowledge of how individual algorithms work.

One parameter, which is independent of the training algorithm, can however be tweaked rather easily – the steepness of the activation function. Activation function is the function that determines when the output should be close to 0 and when it should be close to 1, and the steepness of this function determines how soft or hard the transition from 0 to 1 should be. If the steepness is set to a high value, the training algorithm will converge faster to the extreme values of 0 and 1, which will make training faster for an *e.g.* the language classification problem. However, if the steepness is set to a low value, it is easier to train an ANN that requires fractional output, like *e.g.* an ANN that should be trained to find the direction of a line in an image. For setting the steepness of the activation function FANN provides two functions: `fann_set_activation_steepness_hidden` and `fann_set_activation_steepness_output`. There are functions because it is often desirable to have different steepness for the hidden layers and for the output layer.

FANN possibilities

The language identification problem belongs to a special kind of function approximation problems known as classification problems. Classification problems have one output neuron per classification and in each training pattern precisely one of these outputs must be 1. A more general function approximation problem is where the outputs are fractional values. This could *e.g.* be approximating the distance to an object viewed by a camera or even the energy consumption of a house. These problems could of course be combined with classification problems, so there could be a classification problem of identifying the kind of object in an image and a problem of approximating the distance to the object. Often this can be done by a single ANN, but sometimes it might be a good idea to keep the two problems separate and *e.g.* have an ANN which classifies the object and an ANN for each of the different objects which approximates the distance to the object.

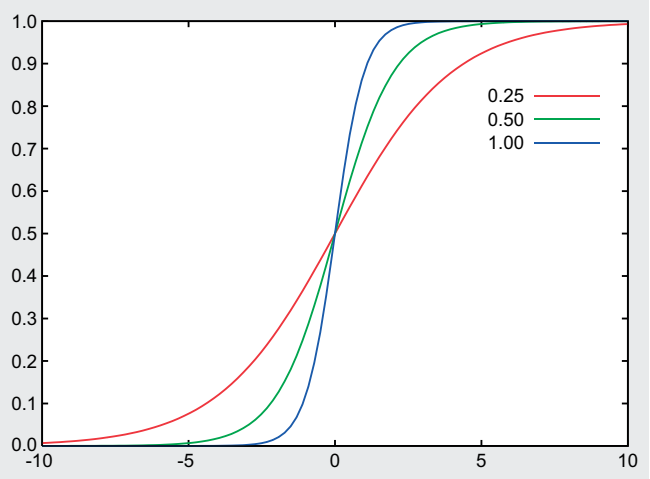

Figure 3. *A graph showing the sigmoid activation function for the steepness of 0.25, 0.50 and 1.00*

Artificial Intelligence

When is something or somebody intelligent? Is a dog intelligent? How about a newborn baby? Normally, we define intelligence as the ability to acquire and apply knowledge, reason deductively and exhibit creativity. If we were to apply the same standards to artificial intelligence (AI), it would follow that there is currently no such thing as AI. Normally, however, AI is defined as the ability to perform functions that are typically associated with the human intelligence. Therefore, AI can be used to describe all computerised efforts dealing with learning or application of human knowledge. This definition allows the AI term to describe even the simplest chess computer or a character in the computer game.

Another kind of approximation problems is time-series problem, approximating functions which evolve over time. A well known time-series problem is predicting how many sunspots there will be in a year by looking at historical data. Normal functions have an x-value as an input and a y-value as an output, and the sunspot problem could also be defined like this, with the year as the x-value and the number of sun spots as the y-value. This has, however, proved not to be the best way of solving such problems. Time-series problems can be approximated by using a period of time as input and then using the next time step as output. If the period is set to 10 years, the ANN could be trained with all the 10-year periods where historical data exists and it could then approximate the number of sunspots in 2005 by using the number of sunspots in 1995 – 2004 as inputs. This approach means that each set of historical data is used in several training patterns, *e.g.* the number of sunspots for 1980 is used in training patterns with 1981 – 1990 as outputs. This approach also means that the number of sunspots cannot be directly approximated for 2010 without first-approximating 2005 – 2009, which in turn will mean that half of the input for calculating 2010 will be approximated data and that the approximation for 2010 will not be as precise as the approximation for 2005. For this reason, time-series prediction is only well-fitted for predicting things in the near future.

Time-series prediction can also be used to introduce memory in controllers for robots etc. This could *e.g.* be done by giving the direction and speed from the last two time steps as input to the third time step, in addition to other inputs from sensors or cameras. The major problem of this approach is, however, that training data can be very difficult to produce since each training pattern must also include history.

FANN tips & tricks

Lots of tricks can be used to make FANN train and execute faster and with greater precision. A simple trick which can be used to make training faster and more precise is to use input

On the Net

- The FANN library
  http://fann.sourceforge.net
- Martin Riedmiller and Heinrich Braun, *A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm*
  http://citeseer.ist.psu.edu/riedmiller93direct.html
- ANN FAQ
  ftp://ftp.sas.com/pub/neural/FAQ.html

and output values in the range -1 to 1 as opposed to 0 to 1. This can be done by changing the values in the training file and using `fann_set_activation_function_hidden` and `fann_set_activation_function_output` to change the activation function to `FANN_SIGMOID_SYMMETRIC`, which has outputs in the range of -1 and 1 instead of 0 and 1. This trick works because 0 values in ANNs have an unfortunate feature that no matter which value the weight has, the output will still be 0. There are of course countermeasures in FANN to prevent this from becoming a big problem; however, this trick has been proved to reduce training time. The `fann_set_activation_function_output` can also be used to change the activation function to the `FANN_LINEAR` activation function which is unbounded and can, therefore, be used to create ANNs with arbitrary outputs.

When training an ANN, it is often difficult to find out how many epochs should be used for training. If too many epochs are used during training, the ANN will not be able to classify the training data. If, however, too many iterations are used, the ANN will be too specialised in the exact values of the training data and the ANN will not be good at classifying data it has not seen during training. For this reason, it is often a good idea to have two sets of training data, one applied during the actual training and one applied to verify the quality of the ANN by testing it on data which have not been seen during the training. The `fann_test_data` function can be used for this purpose, along with other functions which can be used to handle and manipulate training data.

Transforming a problem into a function which can easily be learned by an ANN can be a difficult task, but some general guidelines can be followed:

- Use at least one input/output neuron for each informative unit. In the case of the language classification system, this means to have one input neuron for each letter and one output neuron for each language.
- Represent all the knowledge that you as a programmer have about the problem when choosing the input neurons. If you *e.g.* know that the word length is important for the language classification system, then you should also add an input neuron for the word length (this could also be done by adding an input neuron for the frequency of spaces). Also, if you know that some letters are only used in some languages, then it might be an idea to add an extra input neuron which is 1 if the letter is present in the text and 0 if the letter is not present. In this way even a single Polish letter in a text can help classifying this text. Perhaps you know that some languages contain more vowels than others and you can then represent the frequency of the vowels as an extra input neuron.
- Simplify the problem. If you *e.g.* want to use an ANN for detection of some features in an image, then it might be a good idea to simplify the image in order to make the problem easier to solve, since the raw image will often contain far too much information and it will be difficult for the ANN to filter out the relevant information. In images, simplification can be done by applying some filters to do smoothing, edge-detection, ridge-detection, grey-scaling etc. Other problems can be simplified by preprocessing data in other ways to remove unnecessary information. Simplification can also be done by splitting an ANN into several easier-to-solve problems. In the language classification problem, one ANN could *e.g.* distinguish between European and Asian languages, while two others could be used to classify the individual languages in the two areas.

While training the ANN is often the big time consumer, execution can often be more time-critical – especially in systems where the ANN needs to be executed hundreds of times per second or if the ANN is very large. For this reason, several measures can be applied to make the FANN library execute even faster than it already does. One method is to change the activation function to use a stepwise linear activation function, which is faster to execute, but which is also a bit less precise. It is also a good idea to reduce the number of hidden neurons if possible, since this will reduce the execution time. Another method, only effective on embedded systems without a floating point processor, is to let the FANN library execute by using integers only. The FANN library has a few auxiliary functions allowing the library to be executed using only integers, and on systems which does not have a floating point processor this can give a performance enhancement of more than 5000%.

A tale from the Open Source world

When I first released the FANN library version 1.0 in November 2003 I did not really know what to expect, but I thought that everybody should have the option to use this new library that I had created. Much to my surprise, people actually started downloading and using the library. As months went by, more and more users started using FANN, and the library evolved from being a Linux-only library to supporting most major compilers and operating systems (including MSVC++ and Borland C++). The functionality of the library was also considerably expanded, and many of the users started contributing to the library. Soon the library had bindings for PHP, Python, Delphi and Mathematica and the library also became accepted in the Debian Linux distribution.

My work with FANN and the users of the library takes up some of my spare time, but it is a time that I gladly spend. FANN gives me an opportunity to give something back to the open source community and it gives me a chance to help people, while doing stuff I enjoy.

I cannot say that developing Open Source software is something that all software developers should do, but I will say that it has given me a great deal of satisfaction, so if you think that it might be something for you, then find an Open Source project that you would like to contribute to, and start contributing. Or even better, start your own Open Source project. ■