Spam Image Identification Using an Artificial Neural Network

Jason R. Bowling, Priscilla Hope, Kathy J. Liszka
The University of Akron
Akron, Ohio 44325-4003
{bowling, ph11, liszka}@uakron.edu

Abstract

We propose a method for identifying image spam by training an artificial neural network. A detailed process for preprocessing spam image files is given, followed by a description on how to train an artificial neural network to distinguish between ham and spam. Finally, we exercise the trained network by testing it against unknown images.

1. Introduction

Select – delete – repeat. It's what we spend the first ten minutes of every day doing -- purging spam from our inboxes. In the first month after the National Do Not Call Registry went into affect, we noticed about a 30% increase in spam. No, that's not backed by scientific process, just personal observation. And then, it got worse. Clearly spam isn't going away, at least not in the foreseeable future. People still respond to it, buy products from it, and are scammed by it.

Filters are available to combat these unsolicited nuisances. But spammers continually develop new techniques to avoid detection by filters. See [1] for a current and comprehensive list of spam techniques. This paper focuses on one specific category of unsolicited bulk email – image spam. This is a fairly recent phenomenon that has appeared in the past few years. In 2005, it comprised roughly 1% of all emails, then grew to an estimated 21% by mid 2006 [2]. They come as image attachments that contain text with what looks like a legitimate *subject* and *from* address. They are successfully getting by traditional spam filters and optical character recognition (OCR) systems. As a result, they are often referred to as OCR-evading spam images. A common example is shown in Figure 1. These come in many forms by way of file type, multipart images where the image is split into multiple images, and even angled, or twisted.

A number of image spam identification and classification techniques have been proposed [3, 4, 5] including image processing and computer vision techniques [6, 7]. We are studying the use of an artificial neural network (ANN) to identify the difference between a spam image from a "ham" (i.e., non-spam) image. ANNs have been used in [8] to identify spam by looking at the

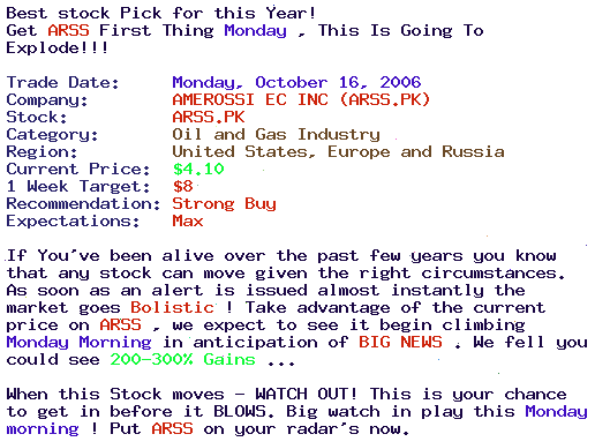

Figure 1. Simple Image Spam

text-based header portion of spam email. In our research, we are interested in spam that uses jpg, bmp, gif and png images.

An artificial neural network (ANN) is a computational model based on biological neural networks. Given proper inputs, they are supposed to be adaptive, learning by example. An ANN is defined by a set of input and output variables and then it is given a set of training examples. The Fast Artificial Neural Network (FANN) [9] is an open source library that implements an ANN. An excellent primer on artificial neural nets and, specifically, the FANN libraries can be found in [10].

The process is accomplished in three steps. First is image preparation. We create a file compatible with the inputs of the ANN we are testing. Next we train the artificial neural network with our training data. Finally, we test the network with "unknown" images to see how well it has learned to identify spam versus ham. Sections 2 through 4 provide a detailed description of how we set up our experiments. Code is included in the appendices for the interested reader. We've provided everything one needs to create and test an ANN except the spam. You'll need to raid your own inbox for that.

2. Image Preparation

The first step in the overall process is to prepare the images in a standard format. We have a small C program called image2fann (see Appendix A) that takes images in most any common format using a utility called ImageMagick [11]. This is an open source utility that converts and formats images from virtually any format to another. In our case, we take gif, bmp and png files, then convert them to jpg and scale them down to a standard size for processing. This is where we also set up the rest of the format for the ANN input file. The program handles multipart images, creating separate lines of data for each frame, or single part images. The process works as follows.

- Use the ImageMagick utility to make a rescaled 150 × 150 pixel jpg image from the source input file.
- Use the ImageMagick utility a second time to convert the 150 × 150 pixel jpg image into a 150 × 150 pixel 8-bit grayscale image. A temporary file is created for the next step.

At this point, the image consists of 150 × 150 = 22,500 values that range from 0 – 255.

- Read in the temp file byte by byte and divide the pixel values by 1000 to scale them between 0 - 1.
- Create pairs of lines that consists of the following:
- The first line of each pair contains output of all 22,500 of the scaled values. (Yes, in a single line.)
- A second line is paired with the image data that represents the training value. We use 1 for spam and -1 for ham.
- Create these line pairs for each image and output to a file. We call this *training.data*.

At this point, we have a file that contains a large number of line pairs that look something like:

```
.128 .123 .156 .128 .156 .254 …
1
.156 .128 .128 .123 .156 .254 …
-1
```

Finally, we need to insert a header line so that our file will be properly FANN-formatted. The format is specific:

<# input sets> <# inputs> < # output nodes in the net>

If we have a file of data for 500 images, the header would look like the following with no blank line between this header and the first line of data,

```
500 22500 1
```

The next step is to write a program to create and train an artificial neural net.

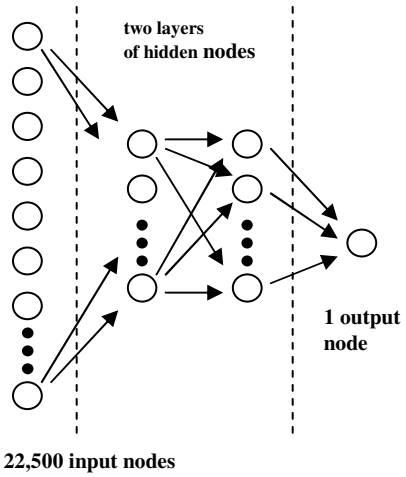

Figure 2. The ANN

3. FANN Training

We selected the FANN libraries because they are open source, the learning curve is not steep, and they claim that the ANN execution time is extremely fast. The training code posted in Appendix B is based on basic FANN provided functions necessary to create, train and run our ANN. The code was developed with the help of examples provided in the FANN documentation.

The first function used is *fann_create_standard* . This library function creates a fully connected backpropagation neural network. This means that we take the human led teaching approach, otherwise known as a supervised learning paradigm. We feed the ANN with training images and explicitly tell it whether the image is ham or spam. The hope is that given enough learning data, the ANN can evolve enough to identify images independently.

Our ANN is created with 22,500 inputs, two hidden layers of 50 or 75 nodes each, and one output node. Figure 2 shows a rough approximation of our network. The input nodes are the pixels of an image. The output layer is the -1 or 1 indicating ham or spam. Determining the number of hidden layers is largely an experimental process. Too few layers and the network cannot learn well. Too many, and it could get into a thrashing situation because it cannot identify the true feature(s) in the input

layers that teach it how to arrive at the desired output value.

The function *fann_read_train_from_file* loads the data file we created in image2fann.cpp (ex. *training.data*). We set values for *fann_set_activation_steepness_hidden* and *fann_set_activation_steepness_output* to 1 for the hidden and output layers respectively. Nodes in an ANN, (also called neurons), are implemented as mathematical functions. An *activation function* takes the inputs to a node, uses a weight for each input and determines the weight of the output from the node. In the FANN libraries, the steepness determines how the activation function behaves. Steepness values over 0.5 are used for classification problems. Smaller (less steep) values are suited for function approximation applications. We use a value of 1 in our application since the purpose of our ANN is to classify images as ham or spam.

Next, the function *fann_init_weights* use Widrow and Nguyen's algorithm [12] to set initial weights for the nodes. This attempts an even distribution of weights across each input node's active region so that the ANN will train faster. While its performance is not guaranteed, we found it to be a good choice.

The way the network trains is by continually adjusting each node's weights so that the output of the ANN matches the output given in the training file. An *epoch* is defined as one cycle where the weights are adjusted to match the output in the training file. This is another area where a balance must be reached. Too many epochs can cause the ANN to fail. We use *fann_set_learning_rate* to determine how aggressive the learning algorithm should be. Using the *fann_train_on_data* function, we train to a desired error, stepping down the training rate at preset intervals to avoid oscillation.

The function *fann_run* takes a single image through the network and outputs a single value, as we only use one node in the output layer. This value ranges between -1 and 1.

When it's finished with all of the images, we output a summary of the ANN's results on the training data. Finally we use *fann_save, fann_destroy_train*, and *fann_destroy* to free allocated memory and save our ANN for later use in testing data.

4. FANN Testing

Figure 3 shows the entire process for our experiments, including the final testing step. The code for testing is presented in Appendix C. We wrote a simple bash shell script to format the images for both training and testing.

The function *fann_test* runs an image through the ANN, similar to *fann_train*, except that it does not alter the ANN. No learning is taking place, no modifications

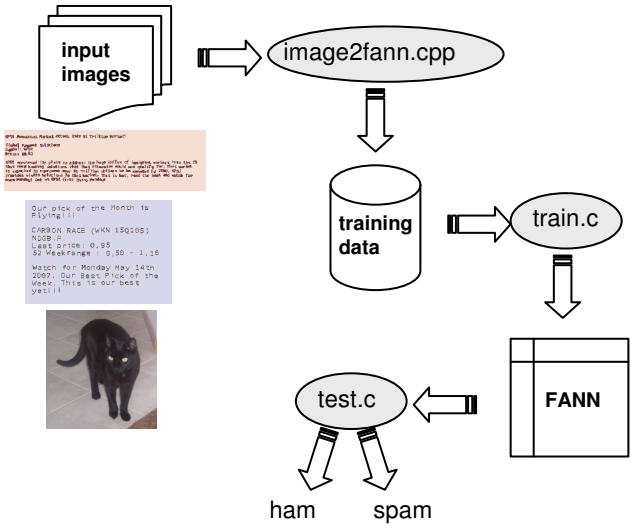

Figure 3. Preparing, Training and Testing

are made to adapt to the image. A value called MSE, or mean square error, is calculated during both training and testing. While this value does not change the state of the network, it can change and should be reset for each image as it is tested. The function *fann_reset_MSE* does this for us.

4.1 Sample Training Results

The following data is a partial output file from a run of the train.c program. Bit fail is the number of output nodes that differ more than the bit fail limit. This is the maximum acceptable difference between the output we are hoping for and the actual output during training. For example, in the first epoch, we set the desired error to 0.4. During that portion of the training, 56 nodes exceeded that limit.

Sample output from train.c (partial)

22604 nodes in network
Max epochs 200. Desired error: 0.4
Epochs 1. Current error: 0.2800000012. Bit fail 56.

Learning rate is: 0.500000

Max epochs 5000. Desired error: 0.2000000030.
Epochs 1. Current error: 0.2800000012. Bit fail 56.
Epochs 20. Current error: 0.2800000012. Bit fail 56.
Epochs 40. Current error: 0.2251190692. Bit fail 56.
Epochs 60. Current error: 0.2074941099. Bit fail 65.
Epochs 71. Current error: 0.1479636133. Bit fail 48.

Learning rate is: 0.200000

```
Max epochs     1000. Desired error: 0.1500000060.
Epochs         1.  Current error: 0.1428321898. Bit fail 47.

Learning rate is: 0.100000

Max epochs     5000. Desired error: 0.0020000001.
Epochs         1.  Current error: 0.1743054837. Bit fail 61.
Epochs        20. Current error: 0.1289706826. Bit fail 43.
Epochs        40. Current error: 0.1205255687. Bit fail 36.
Epochs        60. Current error: 0.1443600655. Bit fail 72.
Epochs        80. Current error: 0.0639020428. Bit fail 16.
```

4.2 Sample Testing Results

The following data is a part output file from a run of the test.c program. The first column is the output value from the node in the output layer. The second value is -1 for ham, 1 for spam, and the difference is the value in the last column.

Sample output from test.c (partial)

```
-0.386381  should be -1.000000   difference = 0.613619
-0.988860  should be -1.000000   difference = 0.011140
-0.883410  should be -1.000000   difference = 0.116590
-0.651567  should be -1.000000   difference = 0.348433
-0.999994  should be -1.000000   difference = 0.000006
1.000000   should be -1.000000   difference = 2.000000
-1.000000  should be -1.000000   difference = 0.000000
0.977352,  should be  1.000000   difference = 0.022648
0.999992   should be  1.000000   difference = 0.000008
1.000000   should be  1.000000   difference = 0.000000
0.977336   should be  1.000000   difference = 0.022664
```

4.3 Sample Runs

We ran many trial runs as we accumulated spam, starting with networks trained with as few as 124 images. Here, we've trained our ANN with 572 images, 508 of them spam, and 64 ham. Testing was done with 30 unseen (to the ANN) spam and 23 unseen ham images. Figure 4 [13] shows results of a network using 50 hidden nodes and 75 hidden nodes respectively.

5. Conclusions and Future Work

We feel the initial results are promising. Our current ANNs are being trained with a target of 1000 spam and 1000 ham as we find and generate spam input images. At this point, we should be able to judge the performance of a 50 hidden node network against a 75 hidden node network. We are also working on a series of specialized ANNs that are trained and tested to only one specific file type.

The image2fann utility program converts multipart gifs into up to 5 individual frames and outputs the appropriate lines for each into the data file. We think it would be better for testing new images to look at the results of all the frames and average them since each frame will produce its own score from the network.

Another avenue we plan to explore is the size of the input data. The 150 × 150 dimension was decided upon initially to avoid making the network too large. The modification to the programs is trivial, but the run time of the training and testing will be noticeable. We are also exploring the effect of the number of hidden nodes in train.c. The number of connections, and thus trainability, changes dramatically when you do this.

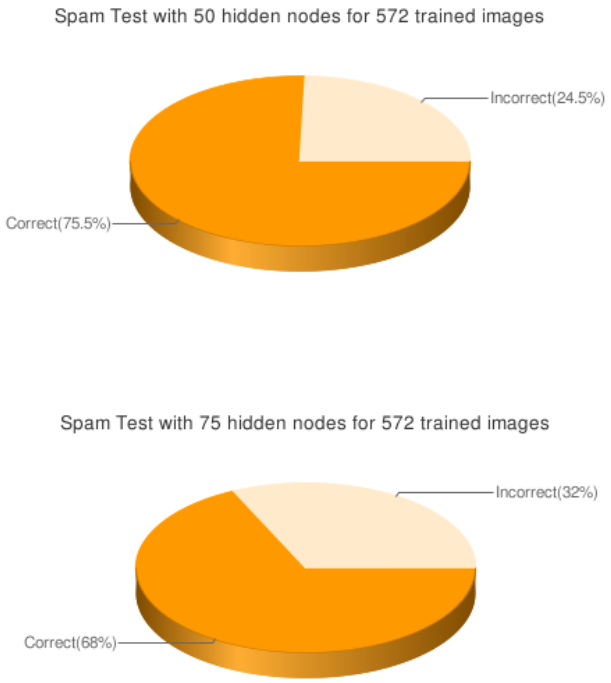

Figure 4. Sample Test Results

6. References

[1] J. Graham-Cumming, The Spammer's Compendium, http://www.jgc.org/tsc.html .

[2] J. Swartz, "Picture this: A sneakier kind of spam," USA Today, Jul. 23, 2006.

[3] H. B. Aradhye, G. K. Myers, and J. A. Herson, "Image analysis for efficient categorization of image-based spam e-mail," *Eighth International Conference on Document Analysis and Recognition*, (ICDAR'05), vol. 2, pp. 914-918, 2005.
[4] C.-T. Wu, K.-T. Cheng, Q. Zhu, and Y.-L. Wu, "Using visual features for anti-spam filtering," *IEEE International Conference on Image Processing*, vol. 3, pp. 509–512, 2005.
[5] M. Dredze, R. Gevaryahu, and A. Elias-Bachrach, "Learning Fast Classifiers for Image Spam," *Fourth Conference on Email and Anti-Spam* (CEAS), 2007.
[6] Z. Wang, W. Josephson, Q. Lv, M. Charikar, and K. Li, "Filtering Image Spam with Near-Duplicate Detection," *Fourth Conference on Email and Anti-Spam* (CEAS), 2007.
[7] G. Fumera, I. Pillai, and F. Roli, "Spam filtering based on the analysis of text information embedded into images," *Journal of Machine Learning Research*, vol. 7, 2699–2720, 2006.
[8] AISK: Artificial Intelligence Spam Killer, http://aisk.tuxland.pl/.
[9] Fast Artificial Neural Network Library, http://leenissen.dk/fann/ .
[10] S. Nissen, "Neural networks made simple," http://leenissen.dk/fann/ .
[11] ImageMagick, http://www.imagemagick.org/script/index.php .
[12] D. Nguyen and B. Widrow, "Reinforcement Learning," Proc. IJCNN, 3, 21-26, 1990.
[13] Google Chart API, http://googlecharter.com

Appendix A

image2fann.cpp

```
/*
      image2fanndata.cpp

      Experimental program to convert
      an image into a 150x150 grayscale
      image, then spit out a FANN-formatted
      training file.

      Insufficient error checking,
      unsafe use of rm via a system call...
      Abandon hope, all ye who enter here.

      DO NOT run this as root. That would be bad.
      DO NOT run on a machine with unbacked up data.

      Author: Jason Bowling
*/

#include "unistd.h"
#include "iostream.h"
#include "stdlib.h"
#include "math.h"
#include "fstream.h"
#include "string.h"

/*
   example of conversion step 1
   convert $1 -resize 150x150! temp-conversion.jpg
*/

int file_exists(char filename[]);
void image2training(char filename[]);

char ifilename[200];
char command_line[300];

int main(int argc, char ** argv) {

   /*
      these flags identify a multipart or single frame input image
   */
   int MULTIPART = 0;
   int SINGLE_FRAME = 0;
   float training_value;

   /*
      make sure we've been passed a filename to work on
   */
   if (argc != 3){
      cout << "Usage: make_input_data <image_file> <desired
                  training_value>\n\n";
      exit(1);
   }

   strcpy(ifilename, argv[1]);
   training_value = atof(argv[2]);
```

```
/*
   Assemble a command line to call ImageMagick's convert utility:

   convert <input image> -resize 150x150! temp-conversion.jpg

   This makes a 150x150 jpeg of input_image, whatever it is.
   Aspect ratio is cheerfully ignored.

   Output will either be the single frame temp-conversion.jpg
   or multipart frames temp-conversion.jpg.0,
   temp-conversion.jpg.1....
*/

strcat(command_line, "convert ");
strcat(command_line, ifilename);
strcat(command_line, " -resize 150x150! temp-conversion.jpg");

system(command_line);

/*
   Figure out what we have.
   Are we a single part image?
*/
if (file_exists("temp-conversion.jpg"))
   SINGLE_FRAME = 1;

/*
   are we a multipart image?
*/
if (file_exists("temp-conversion.jpg.0"))
   MULTIPART = 1;

/*
   sanity check
*/
if ((!MULTIPART) && (!SINGLE_FRAME)){
   //Convert failed, no output files.
   cout << "The call to convert failed.\n\n";
   exit(1);
}

/*
   this should never occur
*/
if (MULTIPART && SINGLE_FRAME){

   cout << "Serious problem: we have evidence "
   cout << " of both single and multipart image.\n\n";
   exit(1);
}

/*
   convert the gray image to a raw 8 bit binary file for easy reading
*/
if (SINGLE_FRAME){
   convert the gray image to a raw 8 bit binary file for easy reading
   system("convert temp-conversion.jpg -depth 8 -size 150x150+0
           gray:out.bin");
   image2training("out.bin");
   cout << training_value << endl;
   system("rm -f temp-conversion.jpg");
```

```
      system("rm -f out.bin");
   }

   /*
      step through the first 5 output frames if they exist;
      convert the gray image to a raw 8 bit binary file for easy reading
   */
   if (MULTIPART){
      if (file_exists("temp-conversion.jpg.0")){
         system("convert temp-conversion.jpg.0 -depth 8 -size 150x150+0
            gray:out.bin");
         image2training("out.bin");
         cout << training_value << endl;
         system("rm -f temp-conversion.jpg.0");
         system("rm -f out.bin");
      }

      if (file_exists("temp-conversion.jpg.1")){
         system("convert temp-conversion.jpg.1 -depth 8 -size 150x150+0
            gray:out.bin");
         image2training("out.bin");
         cout << training_value << endl;
         system("rm -f temp-conversion.jpg.1");
         system("rm -f out.bin");
      }

      if (file_exists("temp-conversion.jpg.2")){
         system("convert temp-conversion.jpg.2 -depth 8 -size 150x150+0
            gray:out.bin");
         image2training("out.bin");
         cout << training_value << endl;
         system("rm -f temp-conversion.jpg.2");
         system("rm -f out.bin");
      }

      if (file_exists("temp-conversion.jpg.3")){
         system("convert temp-conversion.jpg.3 -depth 8 -size 150x150+0
            gray:out.bin");
         image2training("out.bin");
         cout << training_value << endl;
         system("rm -f temp-conversion.jpg.3");
         system("rm -f out.bin");
      }

      if (file_exists("temp-conversion.jpg.4")){
         system("convert temp-conversion.jpg.4 -depth 8 -size 150x150+0
            gray:out.bin");
         image2training("out.bin");
         cout << training_value << endl;
         system("rm -f temp-conversion.jpg.4");
         system("rm -f out.bin");
      }
   } // if MULTIPART
}


/*
   return if file exists, 0 if it doesn't
*/
int file_exists(char filename[])
{
```

```
   ifstream test_name(filename, ios::in | ios::binary);

   if (test_name){
      return 1;
      test_name.close();
   }
   else
      return 0;
}

void image2training(char filename[])
{
   short int input;
   int r;

   ifstream cil_in(filename, ios::in | ios::binary);

   if (!cil_in){
      cout << "Error opening file data. Exiting." << endl;
      exit(1);
   }

   while ( cil_in && !cil_in.eof() ) {
      r = cil_in.get();
      if (r != -1)
         cout << (float) r/1000.00 << " ";
   }

   cout << endl;
   cil_in.close();
}
```

Appendix B

train.c

```
/*
      train.c

      Experimental program to take a
      FANN-formatted training file
      and create an ANN.

      Author: Jason Bowling
*/
#include <stdio.h>
#include "fann.h"

int main()
{
      fann_type *calc_out;
      const unsigned int num_input = 22500;
      const unsigned int num_output = 1;
      const unsigned int num_layers = 4;

      /* this value can be changed to tweak the network */
      const unsigned int num_neurons_hidden = 50;

      const float desired_error = (const float) 0.02;
      const unsigned int max_epochs = 15000;
```

```
const unsigned int epochs_between_reports = 20;
float learning_rate = .5;
struct fann *ann;
struct fann_train_data *data;
int num_neurons = 0;

unsigned int i = 0;
unsigned int decimal_point;

/*    CREATING NETWORK  */
ann = fann_create_standard(num_layers, num_input,
                           num_neurons_hidden,
                           num_neurons_hidden, num_output);

/* reading training data */
data = fann_read_train_from_file("training.data");

fann_set_activation_steepness_hidden(ann, 1);
fann_set_activation_steepness_output(ann, 1);

fann_set_activation_function_hidden(ann, FANN_SIGMOID_SYMMETRIC);
fann_set_activation_function_output(ann, FANN_SIGMOID_SYMMETRIC);

fann_init_weights(ann, data);

/*
      TRAINING NETWORK
      run x epochs at learn rate .y
*/

fann_train_on_data(ann, data, 200, epochs_between_reports, .4);
fann_set_learning_rate(ann, .5);

fann_train_on_data(ann, data,5000, epochs_between_reports, .2);
fann_set_learning_rate(ann, .2);

fann_train_on_data(ann, data,1000, epochs_between_reports, .15);
fann_set_learning_rate(ann, .1);

fann_train_on_data(ann, data,5000, epochs_between_reports, .002);

/*    TESTING NETWORK   */

printf("Testing network. %f\n", fann_test_data(ann, data));

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


/*    SAVING NETWORK    */

fann_save(ann, "image_spam.net");

/*    CLEANING UP       */

fann_destroy_train(data);
```

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

Appendix C

test.c

```
/*
	test.c

	Experimental program run a data file of unknown images
	through a trained ANN (created by train.c) and determine
	if the images are ham or spam.

	Note that the data file must be in the same format
	as the data file used to create the ANN.

	Author: Jason Bowling
*/

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

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

	struct fann *ann;
	struct fann_train_data *data;

	ann = fann_create_from_file("image_spam.net");

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

	data = fann_read_train_from_file("test.data");

	for(i = 0; i < fann_length_train_data(data); i++)
	{
		fann_reset_MSE(ann);
		calc_out = fann_test(ann, data->input[i], data->output[i]);

		printf("%f, should be %f, difference=%f\n",
			 calc_out[0], data->output[i][0],
			(float) fann_abs(calc_out[0] - data->output[i][0]));
	}

	fann_destroy_train(data);
	fann_destroy(ann);

	return ret;
}
```