A neural network.

Jeroen Laros

May 21, 2003

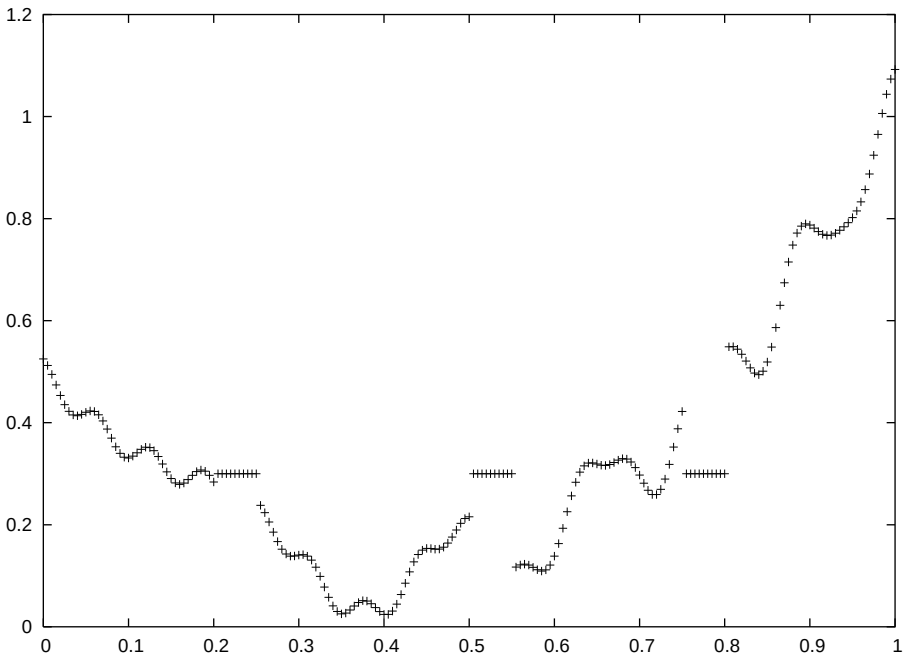
Contents

1 Introduction.

This is a document about an algorithm that tries to find a solution for the following problem: We want to approach a fairly complicated function.

2 Problem.

The function is a real-valued one on a specified domain. We want to build a neural net that approaches this function. This is the function we are going to approach:



3 Approach.

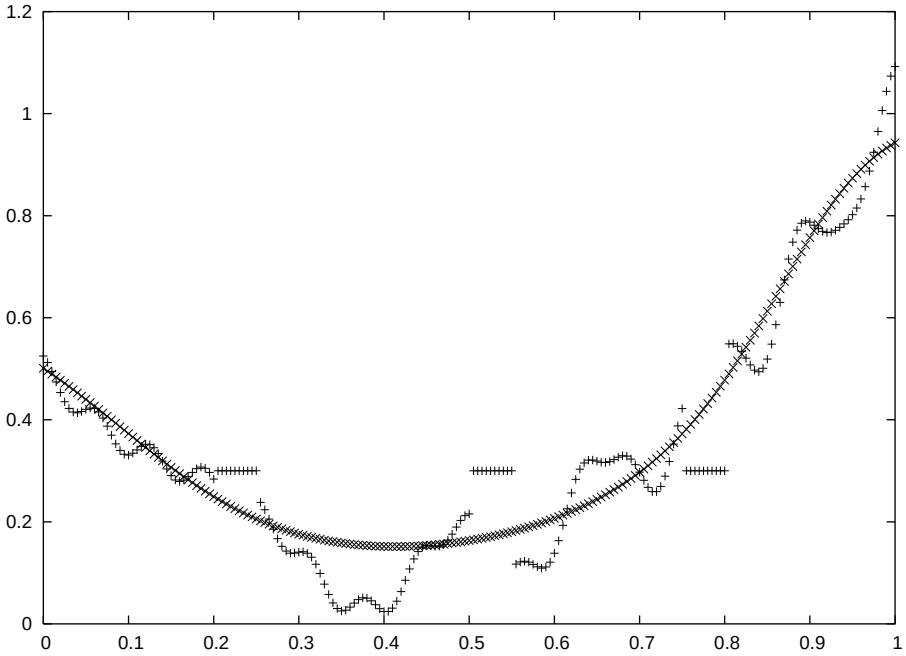
We use a three-layered neural network to approach the function. We use back-propagation as a learning function.

4 Implementation.

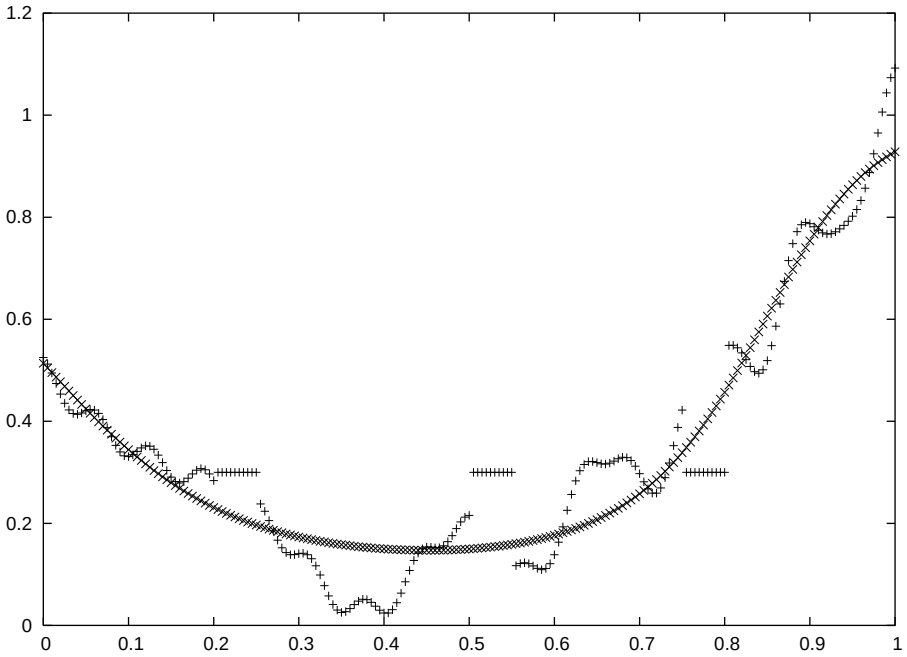
The net itself is a collection of nodes which are connected through edges with weights on them. On each layer (except the output layer) a bias knot is added to simplify the net.

5 Results.

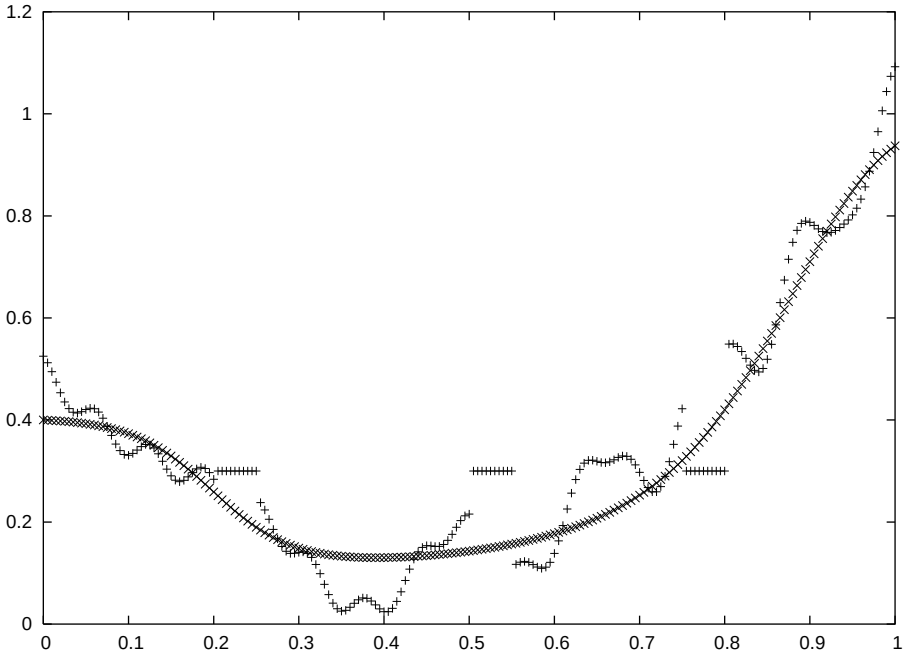
For a neural net with three hidden nodes, 100000 epochs and a learning speed α of 0.9 the output of the trained net looks like this:



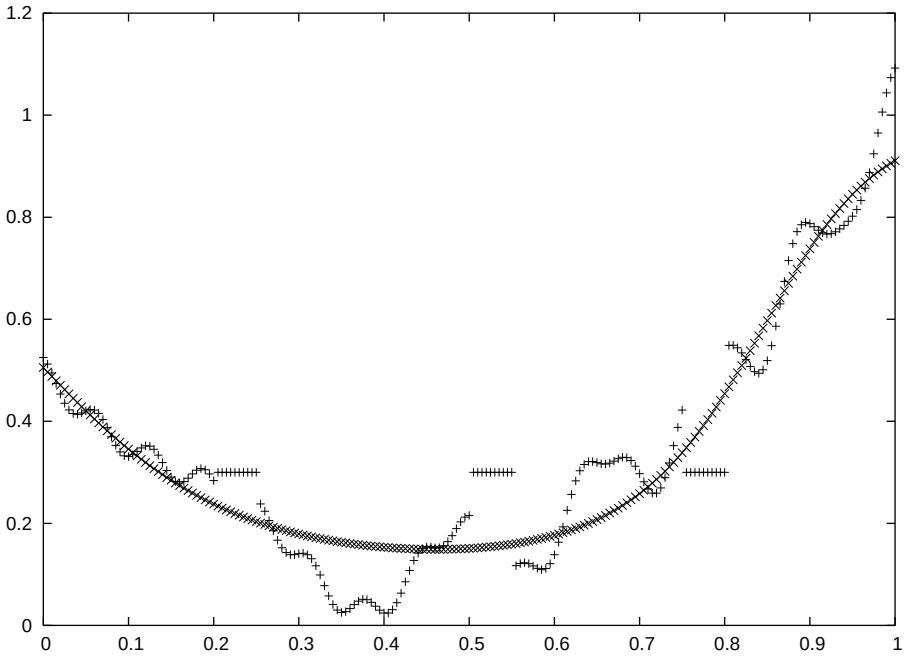
A net with only one node is only capable of making the sigmoid function, so let us look at a net with two nodes, 100000 epochs and an α of 0.9:



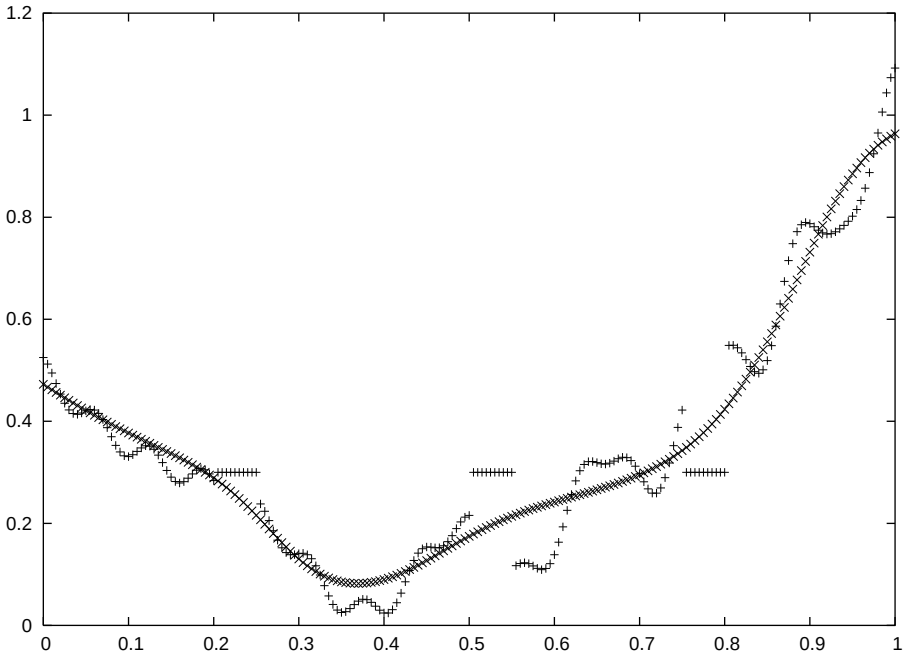
If we increase the number of epochs to 10000000, we get:



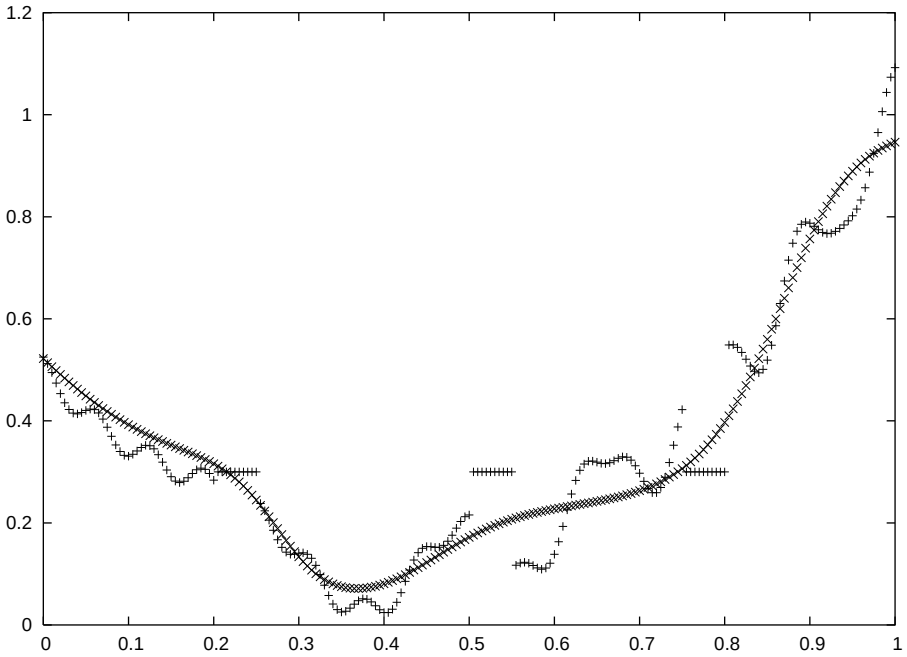
and if we change α to 0.4 we see:



We see that the number of nodes is very important. The overall looks of the function does not change if the number of nodes does not change. So the degree of the function and the number of nodes seems correlated. Now let us look at networks with more nodes. This is the output with four hidden nodes, 1000000 epochs and an α of 0.9:

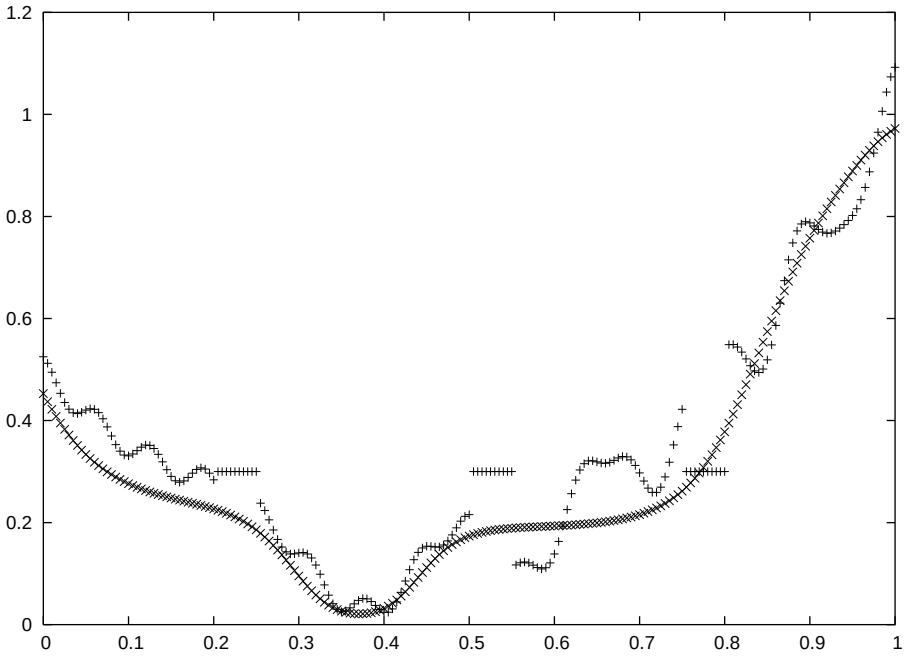


the next one has 40 nodes:



Now it seems no more progress is made, but the more nodes we use, the more time it takes before all nodes are trained sufficiently. The next picture shows

the same network, but it is trained for 10000000 epochs:



6 Conclusion.

It seems that our function can be approximated almost exactly with a net of about thirty nodes and sufficient time to train it. To give an idea about what is meant with sufficient: the last test took 422 seconds on a Pentium-4 at 1.7 GHz and no real improvement with the previous one can be seen.

7 References.

For the assignment itself visit: http://www.liacs.nl/ kosters/AI/nn.html.

8 Appendix.

The code in this section is just the important part of the code. The rest is just some initialisation.

```
#include <stdio.h>  // printf
#include <stdlib.h> // malloc, free
#include <math.h>   // exp
#include "net.h"
```

```
struct branch {
  struct neuron *in;
  float weight;
};//branch

struct neuron {
  branch *inputs;
  float ins;
  float output;
};//neuron

// Global variables.
int numberofnodes = 0,
    r_max = RAND_MAX / 2;
neuron *end = NULL;

// Private prototypes.
float r_value(void),
      sigmoid(float),    // The threshold function.
      sigmund(float);    // The derivate of the threshold function.
void propagate(neuron *, int); // The propagation function.

// Private functions.
// Return a random value (!= 0) between -1 and 1.
float r_value(void) {
  float temp = 0;

  while (!temp)
    temp = ((float)rand() / r_max) - 1;
  return temp;
}//r_value

// The activation function.
float sigmoid(float x) {
  return 1 / (1 + exp(-x));
}//sigmoid

// The derivate function of the activation function.
float sigmund(float x) {
  return sigmoid(x) * (1 - sigmoid(x));
}//sigmund

// The propagation function.
void propagate(neuron *n, int j) {
  n->ins = 0;
  for (int i = 0; i < j; i++)
```

```
    n->ins += n->inputs[i].weight * n->inputs[i].in->output;
  n->output = sigmoid(n->ins);
}//propagate

// Public functions.
// Make a net with one input and one output node and a variable amount
// of hidden nodes.
void buildnet(int nodes) {
  neuron *start = (neuron *)malloc(sizeof(neuron)),
         *bias0 = (neuron *)malloc(sizeof(neuron)), // Bias node on level 0.
         *bias1 = (neuron *)malloc(sizeof(neuron)); // Bias node on level 1.

  end = (neuron *)malloc(sizeof(neuron));
  numberofnodes = nodes + 1;
  end->inputs = (branch *)malloc(numberofnodes * sizeof(branch));
  end->inputs[0].in = bias1;
  end->inputs[0].weight = r_value();
  for (int i = 1; i < numberofnodes; i++) {
    end->inputs[i].in = (neuron *)malloc(sizeof(neuron));
    end->inputs[i].in->inputs = (branch *)malloc(2 * sizeof(branch));
    end->inputs[i].weight = r_value(); // Output of the hidden layer.

    end->inputs[i].in->inputs[0].in = bias0;
    end->inputs[i].in->inputs[0].weight = r_value(); // Input of hidden layer.
    end->inputs[i].in->inputs[1].in = start;
    end->inputs[i].in->inputs[1].weight = r_value(); // Input of hidden layer.
  }//for
  bias0->output = -1;
  bias1->output = -1;
}//buildnet

// Print the net.
void printnet(void) {
  printf("%i\n", RAND_MAX);
  printf("INPUT - HIDDEN:\n");
  for (int i = 1; i < numberofnodes; i++) {
    printf("0 - %i: %f\n", i + 2, end->inputs[i].in->inputs[0].weight);
    printf("1 - %i: %f\n", i + 2, end->inputs[i].in->inputs[1].weight);
  }//for
  printf("HIDDEN - OUTPUT:\n");
  for (int i = 0; i < numberofnodes; i++)
    printf("%i - %i: %f\n", i + 2, numberofnodes + 2, end->inputs[i].weight);
  printf("\n");
}//printnet

// Train the net with functions f(x) in domain [a, b].
```

```
void trainnet(int epochs, float alpha, float (*f)(float x), float a, float b) {
  int x = 0;
  float di = 0,
        dj = 0,
        i = 0,
        dom = b - a;

  while (x < epochs) {
    i = (dom * ((float)rand() / RAND_MAX)) + a;
    end->inputs[1].in->inputs[1].in->output = i;   // start = input
    end->inputs[1].in->inputs[1].in->ins = i;      // start = input
    for (int j = 1; j < numberofnodes; j++)        // hidden layer
      propagate(end->inputs[j].in, 2);
    propagate(end, numberofnodes);                 // output

    // Now back-propagate the error.
    di = sigmund(end->ins) * (f(i) - end->output);
    for (int j = 0; j < numberofnodes; j++) {
      end->inputs[j].weight += alpha * end->inputs[j].in->output * di;
      if (j) {
        dj = sigmund(end->inputs[j].in->ins) * end->inputs[j].weight * di;
        for (int k = 0; k < 2; k++)
          end->inputs[j].in->inputs[k].weight += (alpha *
            end->inputs[j].in->inputs[k].in->output * dj);
      }//if
    }//for
    x++;
  }//while
}//trainnet

// Make a stroll through the domain [a, b] in the net and the
// original function.
void verify(float (*f)(float x), float a, float b, float step) {
  FILE *out = fopen("net.out", "w");

  for (float i = a; i < b; i += step) {
    end->inputs[1].in->inputs[1].in->output = i; // start = input
    for (int j = 1; j < numberofnodes; j++)      // hidden layer
      propagate(end->inputs[j].in, 2);
    propagate(end, numberofnodes);                 // output
    fprintf(out, "%f %f\n", i, end->output);
  }//for
  fclose(out);
}//verify

// Free memory used by the net.
```

```
void destroynet(void) {
  free(end->inputs[1].in->inputs[1].in); // start
  free(end->inputs[1].in->inputs[0].in); // bias0
  free(end->inputs[0].in);               // bias1
  for (int i = 1; i < numberofnodes; i++) {
    free(end->inputs[i].in->inputs);
    free(end->inputs[i].in);
  }//for
  free(end->inputs);
  free(end);
}//destroynet
```