Cubic Spline as Transfer Function in Artificial Neural Networks: Basics and Potential

Hofbauer Heinz
hhofbaue@cosy.sbg.ac.at

Berka Tobias
tberka@cosy.sbg.ac.at

Stütz Thomas
tstuetz@cosy.sbg.ac.at

Abstract

Quite a piece of research has been done recently in the field of activation functions. Two of those are especially noteworthy:
Sopena, Romero and Alquézar showed that periodic activation functions (as opposed to the traditional monotonic functions) are much better at learning classification problems like XOR, Two-Quadrants or Two-Spirals.
On the other hand a local research group found that, if an ANN is evolved, evolving activation functions (by using tables of supported function) and topology yields far better results than ordinary evolution of topology.
These results combined lead to the conclusion that a not necessarily monotonic fully evolvable activation function is supperior in fitness. The use of cubic splines fits these parameters and a few more:
The ability to create cubic splines from tabulated functions (cubic spline interpolation) yields the possibility to evolve the tabulated function and removes the strain to use timeconsuming mathematical calculation during evolution.
Cubic splines have a smooth first and continuous second derivative, thus they are suitable for the standard learing methodes.
This article discusses the implementation and applicability of cubic spline interpolated activation functions.

1 Introduction

1.1 Problem Description

One of the current trends in the field of Artificial Neural Net (ANN) research is the use of activation functions other than the classics, such as sigmoid or heaviside. Periodic activation functions, such as sine, that have long been thought of as unsuitable for ANNs, have proven to lead to results better, or in the worst case equal to the ones achieved with sigmoid activation functions[1]. When trying to solve classic ANN benchmark problems (with an analytical approach) we found that the best results can be found when different activation functions are used.
Another topic that was of interest to our project was the evolvation of ANNs. The Departement of Computer Science at the University of Salzburg runs a research program called "siGis" (which stands for "Salzburg Interest Group on Integrated Systems"), that uses a system for evolving the structure of ANNs (NetGEN)[2] .
When taking these two developments into account, it seems clear that an evolvable activation function could lead to an improvement in ANN performance.
The first approach of siGis was to use function tables, which completely changes the function when mutated. For this project, we chose an approach which makes a more subtle way of changing the activation function possible. We interpolate the activation function from an evolvable point set, using the cubic spline interpolation method, which fulfills the criteria of an activation function in the SNNS[3], being smooth in the first derivative, and continuous in the second derivative, both within the interval and at its boundaries.

1.2 What is the Activation Function?

ANNs consist of two rough components: "neurons" (called units) and the connections between them. A single unit's input is the sum of all weighted inputs. Mathematically expressed this means $inp = \sum_i x_i w_i$, x_i being the input values and w_i the respective weights.
The units output (on all outgoing connections) is $f(inp)$. This function f is the "activation function". The fitness of a net can be measured in the squared standard error (SSE), which is calculated as follows:
t . . . target output
a . . . actual output
n . . . number of test patterns

[1]see [2] and section 2
[2]for more detail see "http://www.cosy.sbg.ac.at/project/sigis/sigis.html"
[3]Stuttgart Neural Net Simulator

$$SSE = \sum_{i=1}^{n} (t_i - a_i)^2$$

1.3 Fitness of Different Activation Functions

More fit activation functions lead to a drastic decrease of the ANN's complexity, which leads to much shorter learning times. Let us take a look at the XOR-Problem to illustrate this:

x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

Table 1: XOR function table

The given set is to be split into two classes by an ANN. By simply taking a look at the problem's graph it becomes evident that no linear function can find an adequate solution for this problem.

Furthermore, even ANNs with monotonic activation function without a hidden layer cannot solve it. We hence decided to use the sine function in the output layer. If the correct weights ($\frac{\pi}{2}$) are to be found by supervised learning, it is only possible with positive initial weights, otherwise the learning algorithm is trapped in local minima. If a customized cubic spline function is used, the standard squared error is below 0.01 after only 15 steps.

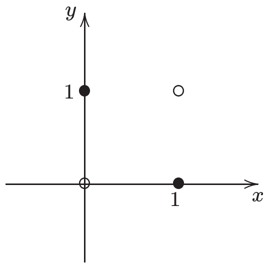

Figure 1: XOR function graph

2 Preliminary Literature

For the readers convenience we include a summary of the article *Neural Networks with Periodic and Monotonic Activation Function: a Comparative Study in Classification Problems* [2].

2.1 Introduction

The article discusses a number of reasons why the use of sine activation functions can lead to a significant improvement in performance of multilayer feed-forward networks.

A classification system assigns a paricular class to each set of values of variables (the input space). A learning algorithm leads a system from an initial state to a minimum of error. Backpropagation is most frequently used.

To improve performance of neural networks several modifications of backpropagation have been proposed, but none of these proposals have considered modifying the activation function. Recent experiments with periodic activation functions (sine) have shown that they are superior in learning speed, computational capacity and generalization.

2.2 Characteristics of Periodic Activation Functions

The use of a periodic function as a classifier divides the input space into an infinitive number of regions, opposed to the two regions created by monotonic activation functions.

Consider the formula

$$g(\vec{x}) = bias_o + \sum_{i=1}^{n_{hid}} \alpha_i sin(\vec{x}\vec{\omega}_i + bias_i)$$

as a representative sinusoid function. The weights ($\vec{\omega}_i$) are called non-linear parameters. The weights of the connections between layers (α_i) are called linear parameters. The biases can be considered as weights with a simple transformation.

This allows approximation of more complex functions. Training a neural network of this kind is basically constructing a Fourier series, the only difference being the possibility of the network to adjust the values of the non-linear parameters.

Networks with sine functions have much more local minima in the error surface, in which the error correction algorithm can be easily trapped before reaching the global minimum, making sufficient variation from the initial position of the weights difficult. This probability can be diminished by assigning an initial set of weights that is appropriate for the problem in question.

Non-linear parameters are difficult to adjust, thus they evolve much more slowly than linear parameters. To enhance the performance it proved to be convenient to assign different values of learning raters for each layer.

2.3 Experiments

Learning speed is measured in the number of epochs required to learn the training set. The learning was considered to be complete when the examples had been solved up to a given tolerance. Partly another criterion has been applied, when learning all examples was impossible or not convenient for generalization purposes. The architecture used was feed-forward with a layer of hidden units. The activation function was sinusoidal, the initial

weights were assigned randomly from a particular interval.
Momentum was set to 0. Without a bias, the network can only produce odd-like functions. A possible approach to the conversion of the target function into an odd-like function is to rescale the variables of the input space between 0 and 1.

2.3.1 Two-Spirals Problem

The points of two interlocking spirals are to be identified. The training set comprises 194 points.
Standard BP with simple structure fails at this problem. The best algorithm ("Cascade Correlation") using standard BP with complex architecture required 1,700 epochs to learn this problem, an algorithm, using sine as the activation function in the hidden layer and hyperbolic tangent in the output layer, only an average of 42.3 epochs. The range of the initial weights is important. With small ranges learning is not possible.

2.3.2 Separation Problem

In this problem, like in the "two spirals" problem, the standard sigmoid function with BP is trapped in local minima. Using sines instead of sigmoids not only prevents the ANN from being trapped, but also allows it to learn around 180 times faster.

2.3.3 Quadrants

All points in quadrant one and three have target 1, the others have target 0. This problem seems to be well suited for sigmoid activation functions, but a 2-2-1 architecture with sine as activation function in the hidden layer and sigmoid in the output layer learns faster than sigmoid function network.

2.4 Conclusion

Networks using sine functions are superior in performance to sigmoids (in terms of learning speed, computational capacity and generalization). Tasks impossible to learn for sigmoid functions were easily learned using sine functions. The main problem is finding an initial set of weights in high-dimensional inputs.

3 Our Approach with Cubic Splines

Our introduction to the topic of activation functions was the article on the fitness of periodic vs. monotonic activation functions ([2]). When studying two classic benchmark problems, the XOR and Two-Quardants Problem, using a sine, and in the Two-Quadrants Problem a heaviside activation

function, we figured that for an analytical approach, it would be good to have an activation function that can be designed as needed. The most reasonable way of doing this is interpolating a function from a set of points. Furthermore, we aim to make the activation function itself evolvable, using NetGen. NetGen is using the SNNS for ANN training. Since the SNNS needs the first and second derivative, the function has to be contiuous.
The goal of cubic spline interpolation is to get an interpolation formula which is smooth in the first derivative, and continuous in the second derivative, both within an interval and at its boundaries.

3.1 Mathematical Theory of Cubic Splines

Given are n distinct points with $P_i = (x_i, y_i)$, $i = 1, \ldots, n$. We assume that the points are ordered and unique with respect to the first coordinate. Thus these n points define $n-1$ intervalls on the x-axes, denoted by $x \in [x_i, x_{i+1}]$. In each of these intervalls a function $f_i(x), i = 1, \ldots, n-1$ is defined. Since we assume that f_i is a cubic spline we may write

$$\begin{aligned} f_i(x) &= f_i(x_i) + a_i \times (x - x_i) + b_i \times (x - x_i)^2 + c_i \times (x - x_i)^3, \\ & \quad x_i \leq x \leq x_{i+1}, i = 1, \ldots, n-1 \end{aligned}$$

For x_{i+1}

$$f_i(x_{i+1}) = f_i(x_i) + a_i \times (x_{i+1} - x_i) + b_i \times (x_{i+1} - x_i)^2 + c_i \times (x_{i+1} - x_i)^3$$

if $\Delta x_i = x_{i+1} - x_i, i = 1, \ldots, n-1$ then

$$f_i(x_{i+1}) = f_i(x_i) + a_i \times \Delta x_i + b_i \times \Delta x_i^2 + c_i \times \Delta x_i^3$$

With another substitution we achieve

$$D_i = \frac{f_i(x_{i+1}) - f_i(x_i)}{\Delta x_i} = a_i + b_i \times \Delta x_i + c_i \times \Delta x_i^2$$

Or first requirements states that the first derivatives have to be equal in $x_i, i = 1, \ldots, n-1$, which yields

$$f'_i(x_{i+1}) = f'_{i+1}(x_{i+1})$$

which means

$$a_i + 2b_i \times \Delta x_i + 3c_i \times \Delta x_i^2 = a_{i+1}$$

Substituting equation in the equation above gives

$$c_i = \frac{a_i + a_{i+1} - 2D_i}{\Delta x_i^2}$$

Our second requirements states that the second derivatives have to be equal in $x_i, i = 1, \ldots, n-1$, which yields

$$f_i''(x_{i+1}) = f_{i+1}''(x_{i+1})$$

which means

$$b_i + 3c_i \times \Delta x_i = b_{i+1}$$

Substituting equation in the equation above gives

$$\frac{D_i - a_i}{\Delta x_i} + 2c_i \times \Delta x_i = \frac{D_{i+1} - a_{i+1}}{\Delta x_{i+1}} - c_{i+1} \times \Delta x_{i+1}$$

Combining equation and the equation above we get

$$\begin{aligned} a_i \times \Delta x_{i+1} + 2a_{i+1}(\Delta x_i + \Delta x_{i+1}) + a_{i+2} \times \Delta x_i &= 3(D_i \times \Delta x_{i+1} + D_{i+1} \times \Delta x_i) \\ & \quad i = 1, \ldots, n-2 \end{aligned}$$

3.2 Implementation of the Cubic Spline Activation Function

The cubic coefficients for each interval are precalculated and stored in a file. The fastest way would be to load these coefficients when the training begins, but this would have been a very drastic change in the SNNS' source code and would have required deep knowledge of the entire code.
Since this was not possible, we decided to read the coefficients from the file when the value of the function is calculated. The calling unit is identified by its name, which is a direct reference of the filename of the function parameter file.
The function we added to the SNNS first calculates the sum of the weighted inputs, and then simply steps through the file, looking for the correct interval. If the unit's input is outside of the interval of the function, it returns the function value at the respective boundary. If within the interval, the corresponding cubic coefficients are loaded and the function value, or the value of the requested derivative, is calculated and returned.
The appendix contains a detailed description of the implementation, along with the source code.

3.3 Results

We tested the cubic spline activation function with the XOR and Two-Quadrants Problem.

3.3.1 XOR-Problem

For a detailed description of this problem see section 1.3.
The ANNs we constructed for the problem had two input units, one output

and no hidden layer. We will now compare three possible solutions (activation function of the input neurons is always the identity function). The activation functions in the output unit are:

1. the SNNS' sine function
2. interpolated sine
3. customized cubic spline

The performance in supervised learning was quite different. The SNNS' sine function was not able to learn the correct setting of the weights. The interpolated sine function (interpolated through $(\pi, 0), (\frac{\pi}{2}, -1), (\frac{\pi}{2}, 1), ...$) was able to learn only with positive initial weights, but a standard squared error of 0 was reached after only 40 epochs. The best performance was reached with a customized function depicted in figure 2.

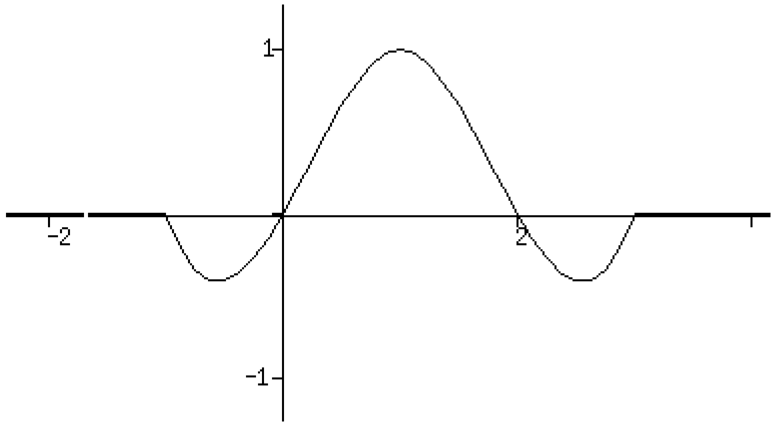

Figure 2: Customized XOR activation function

Using this function, a squared standard error of 0.001 was reached after only 15 epochs, the SSE for these two approaches are compared in figure 3

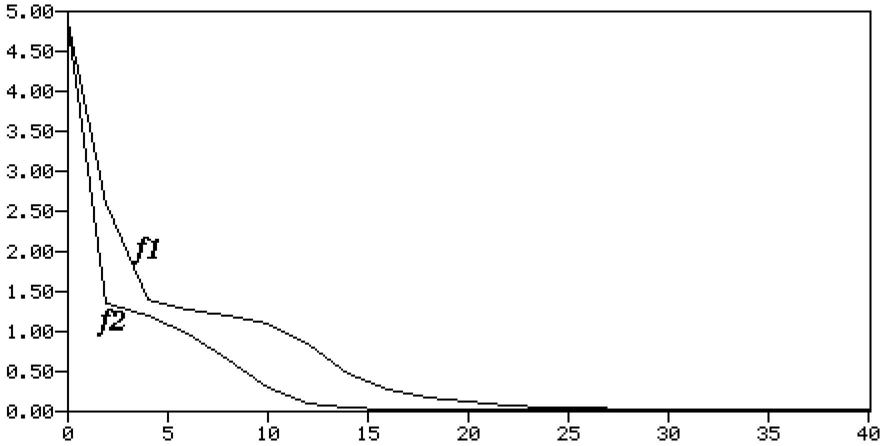

Figure 3: Comparison of learing the XOR Problem with sinus and cubic spline. Both used the same simple architecture network and the same learning pattern, only $f1$ used an interpolated sine as activation function in the output neuron, and $f2$ used the customized cubic spline depicted in figure 2.

3.3.2 Two-Quadrants Problem

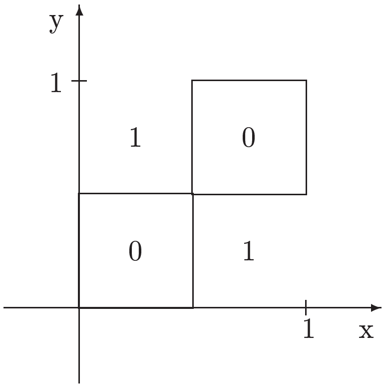

Figure 4: Target output of the Quadrants Problem.

Any point within $([0,1],[0,1])$ should be assigned to either 0 or 1 by an ANN as shown in figure 4.
The activation functions in the hidden layer are interpolated heaviside functions, reducing the problem to the XOR-Problem, which is again solved by the customized cubic spline function in the output layer.
Even though the net works correctly with the interpolated heaviside function when the weights are set to the correct values, learning is hardly ever possible (due to the mass of local minima).

We then experimented with a sine activation function in the hidden layer. With this activation function, learning is possible, but getting the correct weights is still a game of chance.

We obtained the best results with a modification of the interpolated heaviside function. We inclined the two surfaces to give the heaviside a "monotonic touch", which is better for the backpropagation algorithm.

There is still a high chance for the ANN to be trapped in a local minimum, but the overall performance is better than with the two other func-

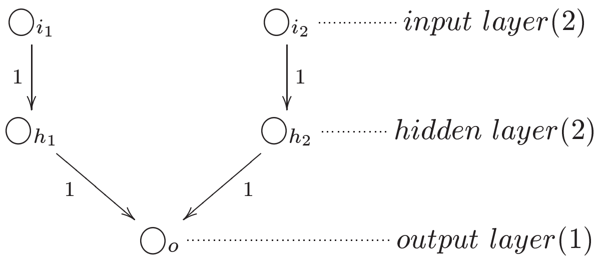

Figure 5: Architecture of the ANN for the Quadrants Problem.

tions. The SNNS' standard way of initializing a net is not very good here, since there is only one interval for both the weights and biases.

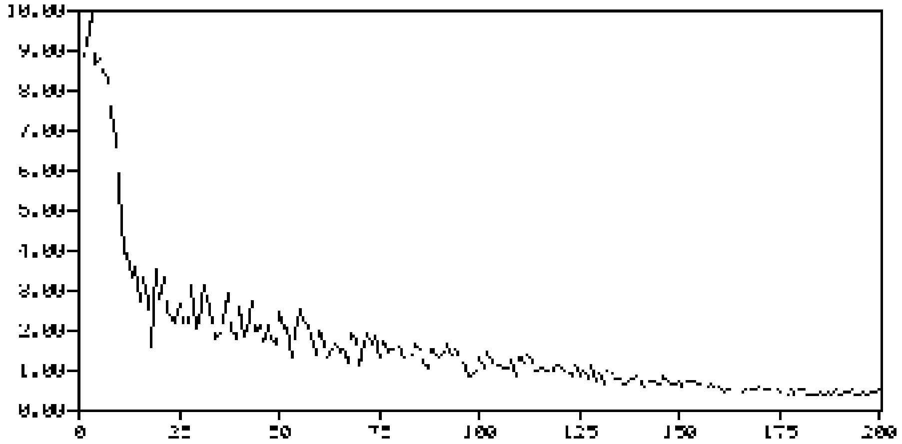

Figure 6: Error surface of quadrants problem using the sine function in the hidden layer.

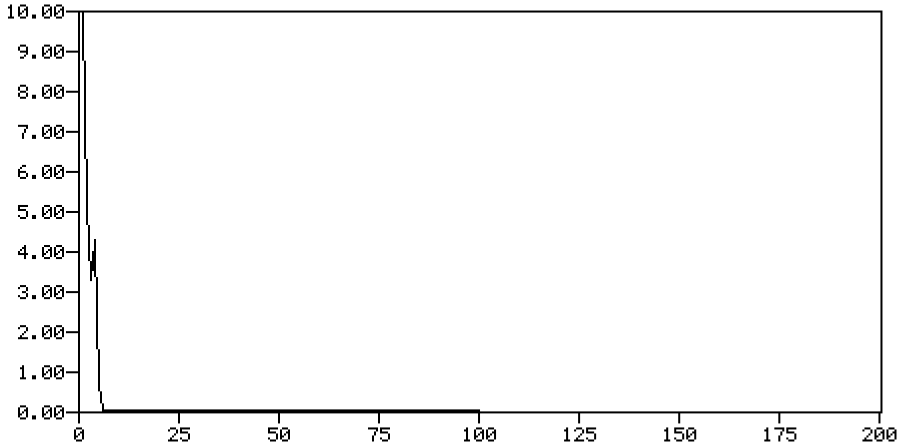

Figure 7: Error surface for quadrants problem using adapted heaviside functions in the hidden layer.

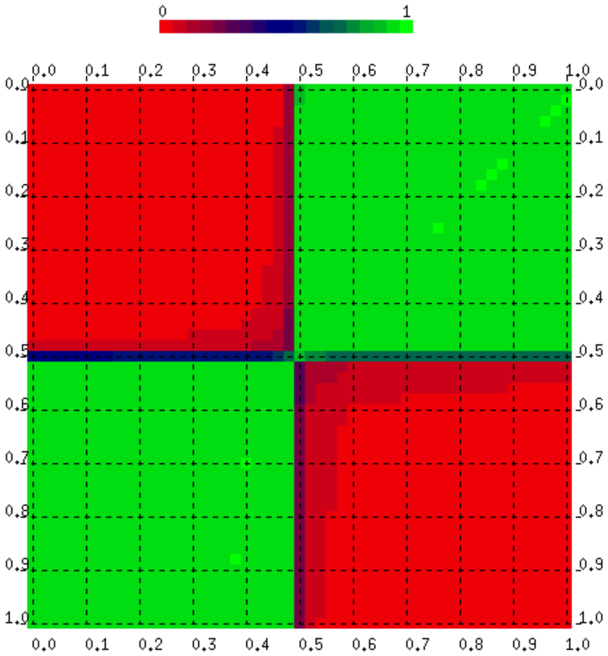

Figure 8: Actual output of the Quadrants Problem using the same architecture as in figure 5 with adapted heaviside as activation fuction.

4 Outlook

When trying to construct ANNs with an analytical approach, cubic spline activation functions have shown to be an efficient tool, but the original idea, and the aim of further research in this field, is to evolve the activation function using evolutionary algorithms.
The research of siGis in the field of ANNs has shown that the use of different activation functions is an excellent way of improving an evolved ANN's performance. The use of activation functions interpolated with the cubic spline method provides a way of evolving the activation function of each unit individually, and an unlimited number of total functions. Furthermore, with cubic spline activation functions the changes to the function can be both small and drastic, depending on how many points are transformed to what degree.
When taking another look at our XOR-Problem example, it is obvious that the essential points (namely $N_1 = (0, 0); N_2 = (1, 1); N_3 = (2, 0)$) can be hit by many different functions, since these points do not necessarily have to be part of the point set from which the function is interpolated.
Taking all this into account, it shall be interesting to test this kind of activation functions in an evolutionary process.

Appendix

A Adding Activation Functions to SNNS in Four Steps

A.1 First Step - Updating the Function Table

As a first step the new activation function has to be entered into the function table of the kernel, located in the file `<SNNS-DIR>/kernel/sources/func_tbl.c`. For one function three entries are necessary:

1. The actual activtaion function.
2. First derivate function of the activation function.
3. Second derivate function of the activation function.

These entries have to be created, even if there is no first or second derivate function.
The three entries for the sine activation function look like this:

1. activation function :
   `{"ACT_Sinus", ACT_FUNC, 0, 0, (FunctionPtr) ACT_SIN},`
2. first derivate function :
   `{"ACT_Sinus", ACT_DERIV_FUNC, 0, 0, (FunctionPtr) ACT_DERIV_SIN},`
3. first derivate function :
   `{"ACT_Sinus", ACT_2_DERIV_FUNC, 0, 0, (FunctionPtr) ACT_2_DERIV_SIN},`

For a better understanding I included the table description of the func_tbl.c file of the SNNSv4.1:

```
/*  The function table contains 5 entries:
     - the XGUI-Identifier of this function (a pointer to a string constant),
       note that the identifier have to match [A-Za-z]^[|, ]* (i.e. the first
       character of the symbol must be a letter and all other characters must
       not match [|, ])

     - an integer constant that determine the class of the function.
       8 different classes are possible:
       OUT_FUNC, ACT_FUNC, ACT_DERIV_FUNC, SITE_FUNC, UPDATE_FUNC, LEARN_FUNC,
       INIT_FUNC, PRUNING_FUNC and TEST_FUNC

     - the number of input parameters for this function (only needed for Update,
       Init, Learning and Test functions)

     - the number of output parameters returned by this function (only needed
```

```
       for Update, Learning and Test functions)

     - a pointer to the (user defined) function.
*/
```

If the activation function does not have a first or second derivate function then the appropriate entry must point to the dummy functions ACT_DERIV_DUMMY or ACT_2_DERIV_DUMMY respectively.

A.2 Second Step - Implementing the Activation Function

The activation functions, and the first and second derivate functions of the activation functions reside in `<SNNS-DIR>/kernel/sources/trans_f.c`. The best idea is probably to copy/paste the declaration from another activation function:

```
FlintType ACT_SIN(struct Unit *unit_ptr)
{
ACT_FUNC_DEFS
register FlintType sum;

}
```

Here it gets a bit vague, as I did not look up how they declared the types‘ used here, but I do have some pretty neat ideas:

FlintType Float.

struct Unit Double linked list with some ID stuff that sparates between site and unit[4].

ACT_FUNC_DEFS Some macros that allow access to the list `struct Unit *unit_ptr`. To see how this macro works either search for them in the SNNS source files or read the code of the other activation functions.

Implementing the sine is quit simple: get the bias, get the inputs for every connection, add them up and return sin(sum).

```
FlintType ACT_SIN(struct Unit *unit_ptr)
{
  ACT_FUNC_DEFS
  register FlintType  sum;
```

[4] Unit is the SNNS expression for neuron

```
  sum =  GET_UNIT_BIAS(unit_ptr);
  if (GET_FIRST_UNIT_LINK( unit_ptr ))
    do
      sum += GET_WEIGHTED_OUTPUT;
    while (GET_NEXT_LINK);
  else
    if (GET_FIRST_SITE( unit_ptr ))
      do
sum += GET_SITE_VALUE;
      while (GET_NEXT_SITE);

  return(sin(SIN_FAKTOR*sum));
}
```

Repeat this process for the first and second derivation function.

A.3 Third Step - Defining the Function Prototypes

Declare the function prototypes in the file `<SNNS-DIR>/kernel/sources/trans_f.h`. In case of the sine activation function and its derivations it looks like this:

```
extern FlintType ACT_SIN(struct Unit *unit_ptr);
extern FlintType ACT_DERIV_SIN(struct Unit *unit_ptr);
extern FlintType ACT_2_DERIV_SIN(struct Unit *unit_ptr);
```

A.4 Fourth Step - Recompiling the Kernel and Xgui

How this is done depends on version and installation, but one of these should work:

- make kernel; make xgui
- make all
- build sim
- build compile

B Implementation of NetGEN Cubic Spline

All files referenced in this section are located in the `<SNNS-DIR>/kernel/sources/` directory.

B.1 func_tbl.c

We added the following to the function table:

```
{"Act_NetGEN_CubicSpline", ACT_FUNC, 0, 0, (FunctionPtr) NetGEN_CS},
{"Act_NetGEN_CubicSpline", ACT_DERIV_FUNC, 0, 0, (FunctionPtr) NetGEN_DERIV_CS},
{"Act_NetGEN_CubicSpline", ACT_2_DERIV_FUNC, 0, 0, (FunctionPtr) NetGEN_2_DERIV_CS},
```

B.2 trans_f.c

We added the function bodies of the NetGEN CubicSpline functions.

We also added a general function to get a FILE pointer to the files with the general function parameters (gfp). This functions was done with respect to future additions of other functions that require gfp.

Additionally we added a function that reads the coresponding file and calculates the spline function in the correct intervall or the requested derivation of the spline.

B.3 trans_f.h

We added the declarations for NetGEN and sine second derivation functions.

B.4 func_mac.h

We added the following macro to get the current unit's name:

```
#define  GET_UNIT_NAME( u_ptr )    ((u_ptr)->unit_name)
```

return value of this function is a `char *` .

B.5 Location and Layout of the GFP-Files

GFP stands for general function parameters. For every type of function used there is an individual directory, thus allowing future additions. The following two subsections deal with the GFP in regards to the cubic spline function.

B.5.1 GFP-Path

The path to the GFP files is given either by the environmental variable SNNS_GFP or, when it is not defined, a subdirectory `.SNNS_GFP` in the user's homedirectory (as defined by the envrionmental variable HOME) is used.
In case of the cubic spline the files are located in a subdirectory /CS.

e.g.: When `SNNS_GFP=/usr/share/snns-gfp` the files are located in the directory `/usr/share/snns-gfp/CS`. If the SNNS_GFP is undefined the HOME variable is used. Assuming that `HOME=/home/stud1/hhofbaue` the the files are located in `/home/stud1/hhofbaue/.SNNS_GFP/CS`.

The file name is identical to the unit name.
More detailed information can be obtained from the source code (section B.6.1).

B.5.2 GFP File Layout

The first entry in the file is the number of entries. Followed by the given number of entries of the following format:

```
struct cs_gfp{
  float x;
  float y;
  float a;
  float b;
  float c;
}
```

Where the formula of the cubic spline function is

$$f(x') = y + a * z + b * z^2 + c * z^3,\ z = x' - x$$

Thus the file looks like this:

```
  int n
  struct cs_gfp(1)
  struct cs_gfp(2)
  .
  .
  .
  struct cs_gfp(n-1)
  struct cs_gfp(n)
```

The cs_gfp entries are sorted from the highest x to the lowest x (going from right to left in the functions graph), such that

$$cs\_gfp(1).x > cs\_gfp(2).x > \ldots > cs\_gfp(n).x$$

B.6 Function Source Code

For the readers convenience we included the code of the added functions.

B.6.1 Source for get_GFP_file

This function returns a file pointer to the given function and filename.

```
FILE *get_GFP_file(char *func, char *filename)
{
    char buf[1024];
    char *env;
    int home = 0;

    if( (env = getenv("SNNS_GFP")) == NULL)
    {
        if( (env = getenv("HOME")) == NULL)
        {
            fprintf(stderr, "get_GFP_file : neither SNNS_GFP nor
                             HOME is set -> cannot generate GFP
                             path \n");
            return NULL;
        }
        home=1;
    }

    strcpy(buf, env);
    if(home) strcat(buf, "/.SNNS_GFP");

    strcat(buf, "/");
    strcat(buf, func);
    strcat(buf, "/");
    strcat(buf, filename);
    return fopen(buf, "r");
}
```

B.6.2 Source for cs_f

This function calculates the value of the cubic spline (in the given derivative) for a specific unit.
Note that the function maps to the identity transfer function if an error occures.

```
float cs_f( float x, char *name, int derivation)
{
    FILE *fp;
    struct {
```

```
    float x;
    float y;
    float a;
    float b;
    float c;
}gfp;

int rv;
float f_ret;
int entries = 0;

if( derivation < 0 || derivation > 2 )
{
    fprintf(stderr, "cs_f : wrong derivation number (%d) ->
                     remap to identity", derivation);
    return x;
}

if ( (fp = get_GFP_file("CS", name)) == NULL ) return x;

rv = fread( &entries, sizeof(entries), 1, fp);
if ( !rv || (entries < 2))
{
    fclose(fp);
    fprintf(stderr, "cs_f : entry missmatch (%d) ->
                     remap function to identity\n", entries);
    return x; /* no entries or to few enries -> id*/
}

rv = fread( &gfp,sizeof(gfp), 1, fp);
if( !rv )
{
    fclose(fp);
    fprintf(stderr, "cs_f : gfp read error (%s) ->
                     remap function to identity\n", name);
    return x;
}

if( x >= gfp.x )
{
    fclose(fp);
    switch(derivation)
    {
        case 0  : return gfp.y;
        case 1  : return 0;
        case 2  : return 0;
        default : break;
    }
    fprintf(stderr, "cs_f : impossible error (wrong derivative
```

```
                     ???) -> remap to idetity\n");
        return x;
    }
    entries--;

    while(entries > 0 && x < gfp.x )
    {
        rv = fread( &gfp, sizeof(gfp), 1, fp);
        if( !rv )
        {
            fclose(fp);
            fprintf(stderr, "cs_f : gfp read error 2 (%s) ->
                             remap to identity\n", name);
            return x;
        }
        entries--;
    };

    if( (entries == 0) && (x < gfp.x) )
    {
        fclose(fp);
        switch(derivation)
        {
            case 0  : return gfp.y;
            case 1  : return 0;
            case 2  : return 0;
            default : break;
        }
        fprintf(stderr, "cs_f : impossible error 2 (wrong derivative
                     ???) -> remap to identity\n");
        return x;
    }

    fclose(fp);
    f_ret = x - gfp.x;

    switch(derivation)
    {
        case 0 :
            f_ret = gfp.y + gfp.a*f_ret + gfp.b*f_ret*f_ret + gfp.c*f_ret*f_ret*f_ret;
            break;
        case 1 :
            f_ret = gfp.a + 2*gfp.b*f_ret + 3*gfp.c*f_ret*f_ret;
            break;
        case 2 :
            f_ret = 2*gfp.b + 6*gfp.c*f_ret;
            break;
        default :
            fprintf(stderr, "cs_f : impossible error 3 (wrong derivative
```

```
                                  ???) -> remap to identity\n");
            f_ret = x;
    };

    return f_ret;
}
```

References

[1] *SNNS User Manual*, downloadable from the University of Stuttgart at: http://inf.informatik.uni-stuttgart.de:80/ipvr/bv/projekte/snns/snns.html

[2] *Neural Networks with Periodic and Monotonic Activation Function: a Comparative Study in Classification Problems* by Josep M. Sopena, Enrique Romero and René Alqueźar

[3] *Learning and Approximation Capabilities of Adaptive Spline Activation Function Neural Networks* (1998) by Lorenzo Vecci, Francesco Piazza and Aurelio Uncini.
Downloadable from: http://www.elsevier.com/inca/tree/