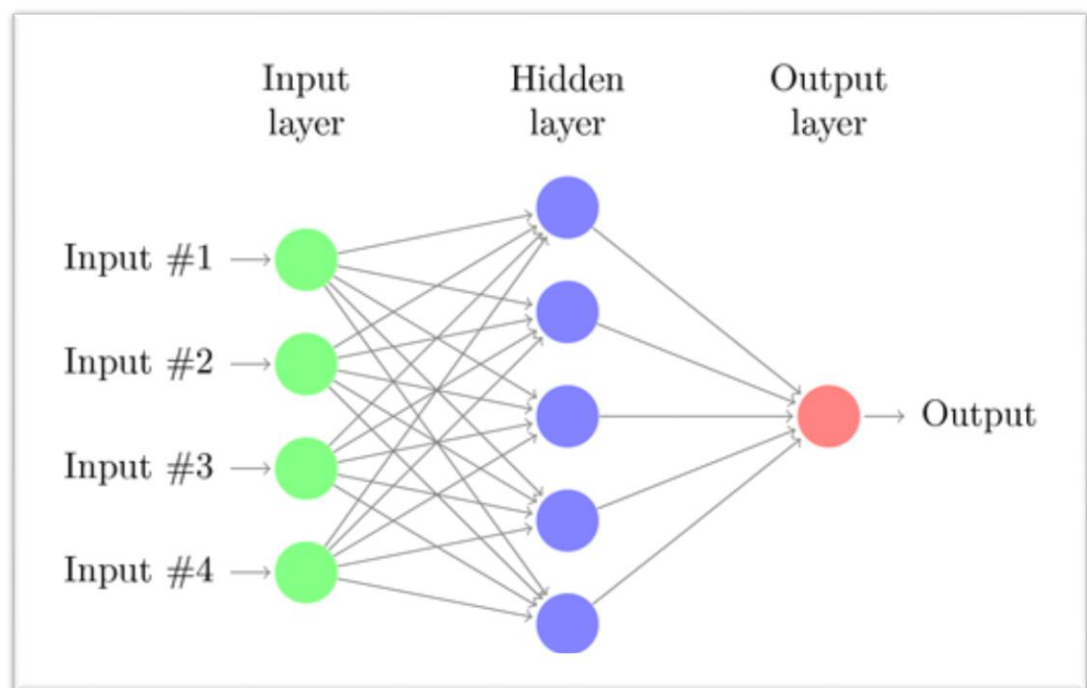


GAME AI REPORT



4/2/2015

MLP versus RBF for function approximation

In this report a continuous function is created and 2 networks are compared to see which is better and to define the characteristics that set them apart. To ensure the networks are compared properly the function will be distorted by minimum noise and the accuracy, speed and computational costs will be compared across each neural network.

Game AI Report

MLP VERSUS RBF FOR FUNCTION APPROXIMATION

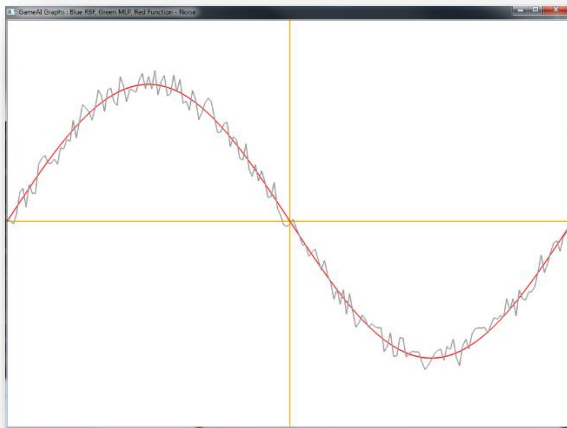
REPORT & PROJECT OVERVIEW

So what is this report... the following is a report based on a comparison of how a multi layered perceptron compares to a radical basis function for continuous function approximation where the function is distorted by noise. For this report a C++ project and a C# project was written establishing an environment in which to compare a MLP to a RBF. As an added plus the project created also outputs graphs on the screen which I will be importing into this very report to ensure that points, assumptions and evidence is provided in a clear and obvious manner.

"I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted." – Alan Turing

THE CONTINUOUS FUNCTION

The function put before both the multi layered perceptron and the radical basis function is as follows. When plotted into our C++ project it produces this curve that is also shown below.



$$y = f(x) + aN$$
$$f(x) = \sin(x)$$

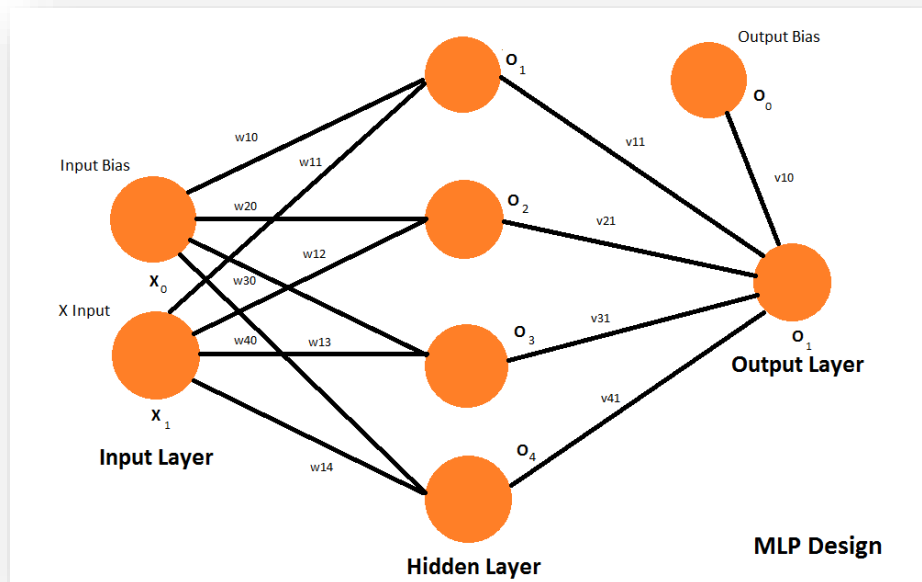
Where x is between 0.0 and 2π and where N is a random floating point value which is treated as noise which is used to distort the graph to ensure the systems designed can successfully approximate functions. Also a is the noise value magnitude which is a floating point number.

DESIGNING SYSTEMS FOR FUNCTION APPROXIMATION

The following is split up into 2 parts. The first being the design and analysis for a multi layered perceptron for function approximation and the second part being the same but for a radial basis function. I will not be covering what these systems are as I have taken the view that the person reading this will indeed know what a multi layered perceptron is as well as a radical basis function.

MULTI LAYERED PERCEPTRON ANALAYSIS & DESIGN

To simplify things here I have created the diagram below to show the design of the multi layered perceptron which will approximate the function which was shown in the first part of this document. I will go over it more in depth in the next section as well.



In the following diagram we have 2 input neurons one for x and the other for bias respectively. We then a single hidden layer which can have a set number of hidden neurons that should be increased when the network needs to be more powerful to approximate the function. In my tests for the continuous function we are using I have found the following value to be optimum for approximating the function.

$$\text{Number of hidden neurons} = 4$$

Finally we have a single output neuron with an additional input neuron which is to act as the output bias. To ensure we can have both a bias for the input and output respectively.

ACTIVATION FUNCTIONS

The activation functions were not covered in the above diagram so this section is dedicated to them. For the activation function in the hidden layer I am using a unipolar sigmoid function rather than the traditional \tanh function. Where the activation values is a linear weighted sum of all the input values.

$$f(act) = \frac{1}{1 + \exp(-act)}$$

As for the output neuron activation function we use a simple linear function for our function approximation task which can be written like below. As we require a linear output for approximation not a nonlinear output.

$$\begin{aligned}act_1 &= w_{10}x_0 + w_{11}x_1 \\act_2 &= w_{20}x_0 + w_{21}x_1 \\&\dots\end{aligned}$$

WEIGHT ADJUSTMENT

For the weight adjustment we also use a momentum term in addition to the back propagation algorithm to ensure weights are adjusted in tune to their contribution to the overall error. The momentum value is an additional coefficient that is added so that the multi layered network can increase the speed of convergence and prevent the system converging to a saddle point.

$$\Delta w_i = M + \eta \delta_i x_i$$

Where M is the momentum value, η the learning rate value, δ_i the error in the network and x_i the value at the current node. It's also worth noting that δ_i changes for the input layer and hidden layer. In the hidden layer we calculate its contribution to the total error where in the input layer we calculate its contribution to the hidden layer as stated by the back propagation algorithm. Calculation of the total error is as follows.

$$\delta_3 = t - o_5$$

However for the input layer we want to calculate the contribution to the hidden layers error rate not the total error and thus this changes to equal the following equation below.

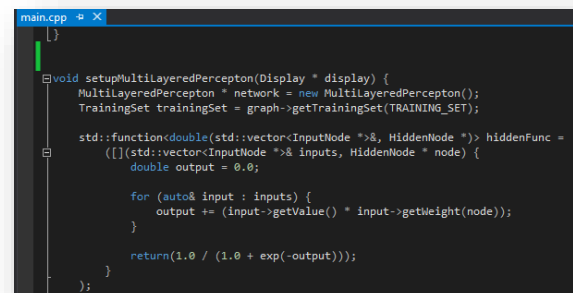
$$\begin{aligned}\delta_1 &= \delta_3 v_{11} o_1 (1 - o_1) \\ \delta_2 &= \delta_3 v_{21} o_2 (1 - o_2) \\ &\dots\end{aligned}$$

EPOCH

The following network also makes use of an epoch value ensuring that we have a maximum number of iterations. Even though we are not training the network by minimum error and instead are using backpropagation with an error tolerance it is still something I have added to ensure we can set the amount of training iterations in the program. If it doesn't train in the given time I use it as a hint to alter the values until the network converges.

TRAINING TYPE

The following network is trained by the epoch value meaning that the network will continue to train until its converged or it exceeds the maximum number of epochs or iterations. In practice I have control over the error threshold and the learning rate and I can adapt them so the network does in most cases converge to a final result. It is also worth noting that all the graphs show in the document have converged.



```
main.cpp
}

void setupMultilayeredPerceptron(Display * display) {
    MultilayeredPerceptron * network = new MultilayeredPerceptron();
    TrainingSet trainingSet = graph->getTrainingSet(TRAINING_SET);

    std::function<double>(std::vector<InputNode *> & inputs, HiddenNode * node) hiddenFunc =
    ([&](std::vector<InputNode *> & inputs, HiddenNode * node) {
        double output = 0.0;

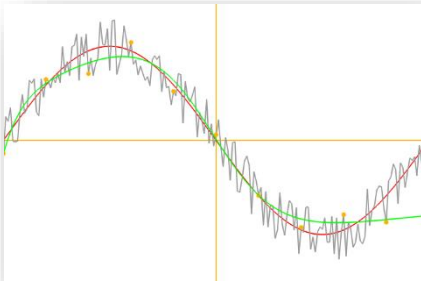
        for (auto& input : inputs) {
            output += (input->getValue() * input->getWeight(node));
        }

        return (1.0 / (1.0 + exp(-output)));
    });
}
```

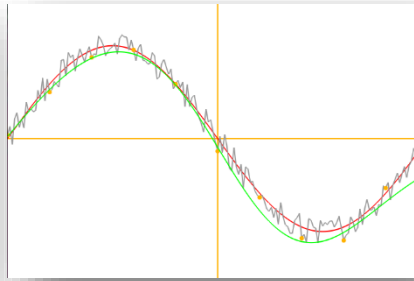
MULTI LAYERED PERCEPTION PERFORMANCE INVESTIGATION

Performance when faced with different noise levels

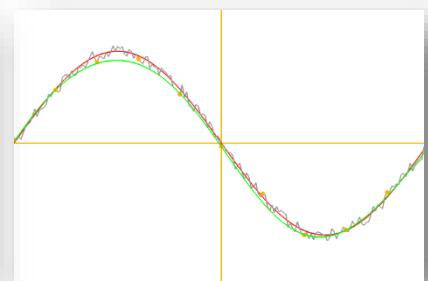
I have generated the following graphs with what I consider to be variations on the level of noise or function distortion in the graph. I have also indicated the value of a in the algorithm below as well. As we can see the multi layered perceptron still approximates the function given a suitable level of noise but you will notice in accuracy when higher noise is applied to the function.



High Noise



Medium Noise



Low Noise

$$y = f(x) + aN$$

$$f(x) = \sin(x)$$

Noise / function distortion levels

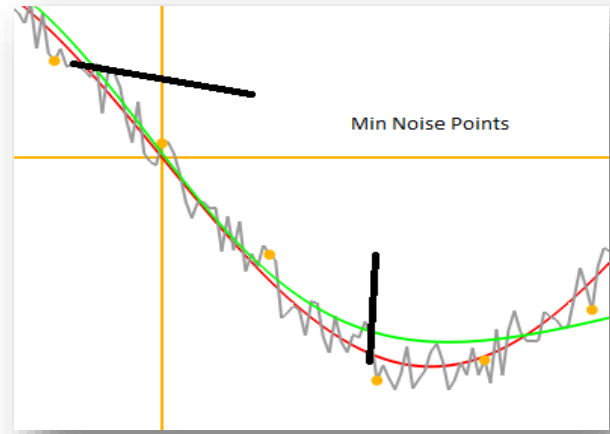
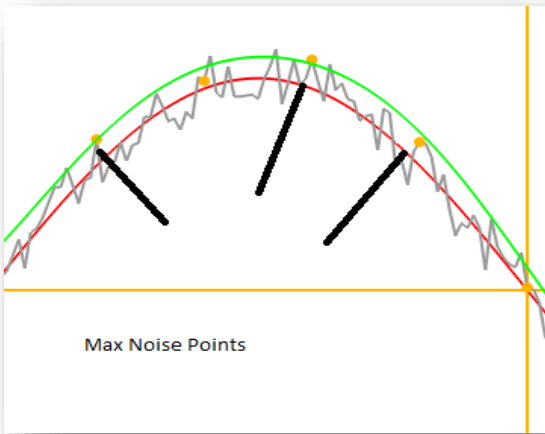
$$\text{High Noise} = 0.2 \quad \text{Medium Noise} = 0.1 \quad \text{Low Noise} = 0.05$$

Following a more detailed analysis I noticed conditions which would alter the performance of the multi layered perceptron. The following observations have been made below.

Larger number of Epochs needed

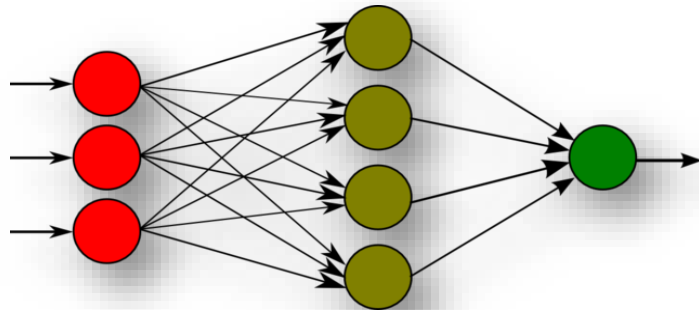
When dealing with a larger amount of noise it is necessary to increase the number of epochs needed to train the network to reach a given error tolerance. However additional Epochs requires longer training times for the inputted data. Additionally having a larger number of Epochs may not always be enough for high levels of distortion. In this case additional hidden neurons would needed to be added to the network resulting in additional weight adjustment and additional values to calculate. Resulting in a more powerful system but a slower system and a system that would potentially over fit the sample data.

Large Amount of Distortion	Medium Amount of Distortion	Low Amount of Distortion
7500 Epochs	2300 Epochs	200 Epochs
Learning Rate = 0.01, Error Tolerance = 0.1	Learning Rate = 0.01, Error Tolerance = 0.1	Learning Rate = 0.01 Error Tolerance = 0.1



Even number of Max & Min Samples

I also noticed that when sampling any distorted function it helped when there was an even distribution of Max and Min noisy data points. If we had a list of graph samples which had only Max noise samples it may approximate the function but the Y value approximated would be too high. The ideal case as shown in the graphs above is when there is an even distribution of noisy data samples so that the network goes between the points producing the correct approximated graph rather than approximating the function plus some amount of noise.



Backpropagation algorithm

I also noted that dealing with a large amount of noise also resulted in the backpropagation algorithm becoming slower still as because the data samples are noisy and less uniform they are in turn harder to approximate. It was also evident that use of the algorithm while having high levels of noise also results in some cases where the network does not fully learn. Moreover the above problems will only ever increase exponentially with each added layer of hidden neurons or additional hidden neurons leading to excessively long training times.

Acceptable Error Tolerance

Another thing I did notice when testing with different error tolerance values is that to properly approximate the function the network should be given an acceptable level of error tolerance. I investigated and found the following value to be the most appropriate in my tests.

$$E = 0.1$$

The reason is so that we make sure that the MLP & RBF doesn't try to map to the exact noise points. We want there to be an acceptable error tolerance as this is an approximation task for the underlying curve rather than the data samples and we want to network to learn the function not the samples.

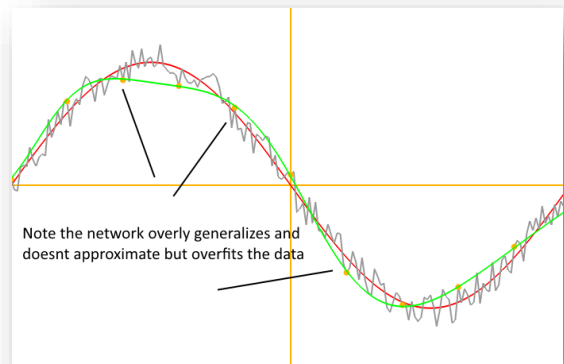
The Number of Hidden Neurons

The investigation of this topic was a bit difficult as I needed to add extra hidden neurons to my multilayered perceptron and then remove one and compare results. Luckily as I had written my own code in an object oriented manner it was easy to add additional hidden neurons to the hidden layer and doing tests. During my tests there were two common cases that would significantly hinder the performance of the network. The first case being dealing with a high number of hidden neurons.

High Number of Hidden Neurons

$$\text{Number of hidden neurons} = 20$$

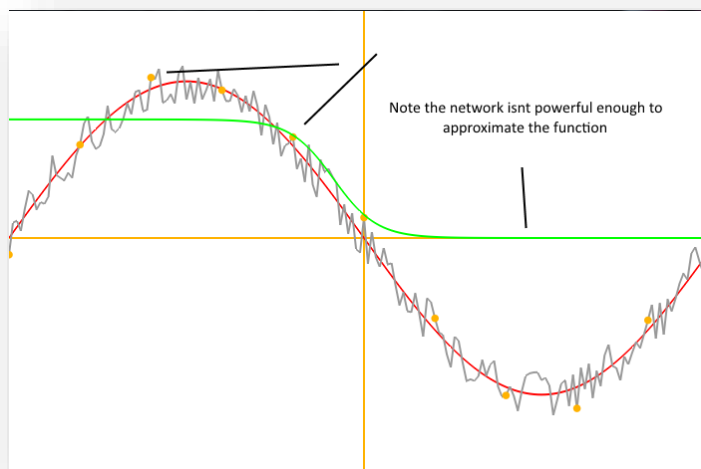
So in my tests where I set the number of hidden neurons to twenty the network tries to fit every data sample sent to it as training data. Even with a stop command when it reaches a set amount of adjustments it's clear that having too many hidden neurons can lead to over fitting and the network will try to fit all data rather than generalize which is what we need for a function approximation problem. What's more it can also lead to increased computational costs as for each neuron added and extra set of weights will need to be updated by the back propagation algorithm in order to get the network to converge. However the reverse situation is just as bad.



Low Number of Hidden Neurons

$$\text{Number of hidden neurons} = 1$$

When I set the number of hidden neurons to 1 I had the new problem. With fewer hidden neurons the network lacks the complexity to be able to be powerful enough to generalize a function however it could still generalize given an extremely high error tolerance rate and an extremely simple function like the following.



$$y = f(x) + aN$$

where $f(x) = 0$

This does come with the nice trade off that it's less computationally expensive however this is not a factor when the network cannot solve the problem it is handed. With such little complexity in the network it can't approximate the vast majority of functions given to it.

Changes with the learning rate value

High Learning Rate Value

$$\eta = 0.3$$

A high learning rate value does have significant advantages. In my tests it became clear that setting the learning rate to 0.3 the network converged quicker in certain circumstances. This can be beneficial if the accuracy of the approximation is not needed to be very accurate. But that is its trade off, that a high learning rate can result in the weights not being able to change in smaller enough numbers to help the network converge to state where its errors are very small.

Low Learning Rate Value

$$\eta = 0.001$$

A low learning rate value for example the one above in my tests was found to be the best factor when wanting to train a network for maximum accuracy. However the network suffered from a very long training times as the network can only increment the weights in small steps even when they are catastrophically wrong. So in reality a medium is required so that the network both converges at a steady pace but also leads to low error rates on the data samples.

Adaptive Learning Rate Mechanism

A happy medium can be found between these two methods by starting with a high learning rate and the decreasing it over time as it converges to its expected result. By doing so we get a faster convergence and a more accurate network. The only drawback is finding out how quickly to reduce the learning rate by which can be best solved through a trial and error approach. In my tests overall I found the below value to be the optimum value for the learning rate which would train the network quickly and accurately.

$$\eta = 0.01$$

Variation of the momentum factor

High Momentum Factor

$$M = 0.1$$

When setting a high momentum factor I noticed problems. First it would occasionally overshoot the minimum and cause the system to become inaccurate and unstable. It also resulted in the network not converging instead hopping back and forth between the minimum. Instead it would only stop after it reached the max amount epochs not the ideal case when training the network.

Low Momentum Factor

$$M = 0.0001$$

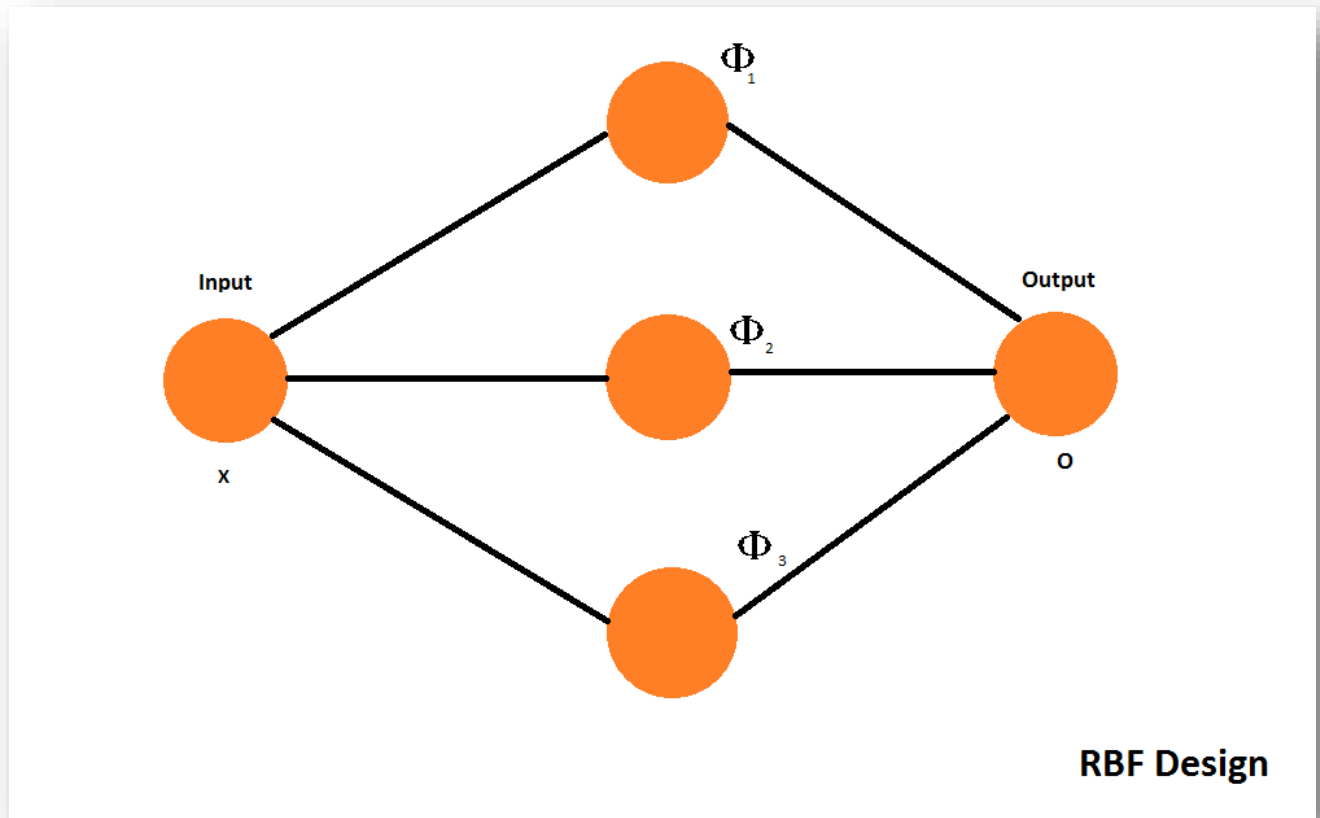
Using a low value instead I found it cannot avoid local minima and never converges to the global minima. It will also add on the effect of slowing down the training of the entire network again another unideal case.

Adaptive Momentum Mechanism

Again a solution exists which is a happy medium between a low and high momentum factor called the adaptive momentum mechanism where the coefficient is made smaller over the course of the training ensuring that the network converges quicker but also doesn't overshoot the minimum error and also retains the advantage of a quicker training time. The optimum value I used in my tests was $M = 0.01$

RADIAL BASIS FUNCTION ANALAYSIS & DESIGN

Again like before I have created a simple diagram to show the design of the radial basis function network below. I will also explain the functions used, positions of the centers and the number of radial basis functions in the hidden layer as well.



Function Used To Map to RBF Space

The most important part of the radial basis function neural network is the function that is used to transform the data from input space to hidden layer space. The function being used to do this is written below.

$$\phi_1(x) = \exp(-\lambda |x - c_1|^2)$$

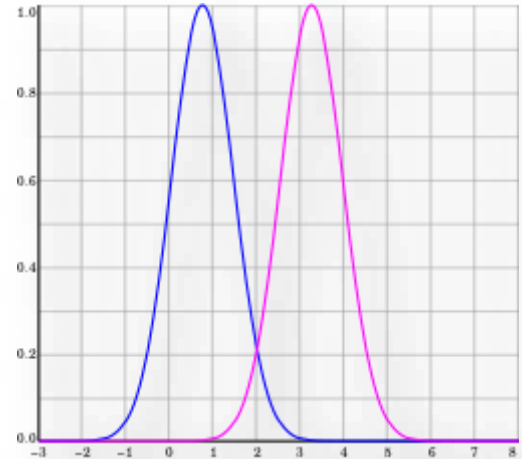
$$\phi_2(x) = \exp(-\lambda |x - c_2|^2)$$

$$\phi_3(x) = \exp(-\lambda |x - c_3|^2)$$

In the following set of equations ϕ is a function which takes a parameter x which will be transformed to hidden layer space. Where λ is the bias term, and c represents the center of the basis function. Centers are distributed evenly across the area being approximated to ensure the radial basis function's width is the same across all centers. Note that we only need to calculate the absolute value between x and c as these aren't 2 or 3 dimensional vectors so to calculate the Euclidian distance we merely need to absolute the difference between them rather than calculate the length of each vector then absolute that difference.

Radial Basis Function Centers

In my radial basis function network I set the centers to be evenly distributed across the area I was approximating with the number of radial basis functions set to be $\frac{3}{5}$ that of the training samples given which during tests was 5, 10 and 20 samples points respectively which resulted in 3, 6 and 12 radial basis functions with their centers distributed evenly across the area of approximation which was from 0 to 2π . The reason for this is because this is an approximation task not a prediction task and therefore we have to allow some error variance as we don't want to map to the noisy data samples but extract the function that is distorted instead.



Error Calculations

Something that the radial basis function has over a multilayered perceptron is that we only have 1 set of weights to calculate the error for which means the error calculation and the weight adjustment is far quicker and results in a much quicker training time. As there is one set of weights we don't need to calculate multiple layers of weights which attribute to a single error. We simply don't use the back propagation algorithm as it's not needed. So we can calculate the error for the weights as follows. Error is abbreviated as δ , output as o and target for the radial basis function as t which in our case is the corresponding y value.

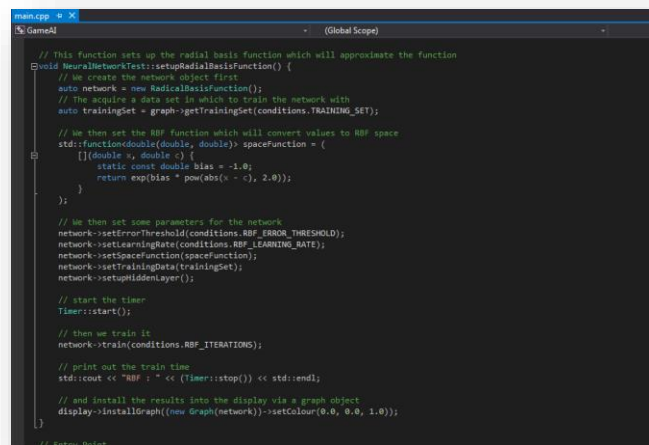
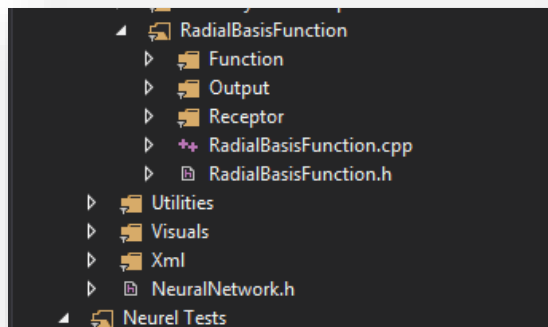
$$o = w_1 \phi_1 + w_2 \phi_2 + w_3 \phi_3$$

$$\delta = t - o$$

Weight Adjustments

As I said above, the radial basis function network has only 1 set of weights instead of multiple layers of weights so updating them is very easy and are updated using the following equation. However we still have an error tolerance for this network which will be used when we want to update the weights.

$$\Delta w_i = \eta \delta \phi_i$$

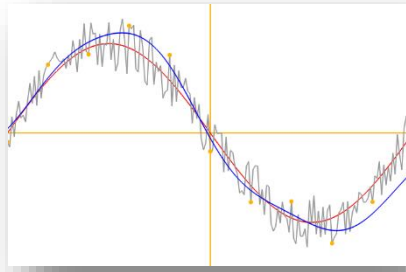


RADIAL BASIS FUNCTION PERFORMANCE ANALYSIS

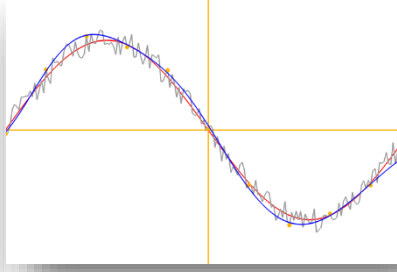
Performance when faced with different noise levels

I have generated the following graphs with what I consider to be variations on the level of noise or function distortion in the graph. I have also indicated the value of a in the algorithm below as well. As we can see the radial basis function approximates the function given a suitable level of noise but unlike the multi layered perceptron you will notice that the accuracy doesn't degrade as fast when it comes to the level of noise.

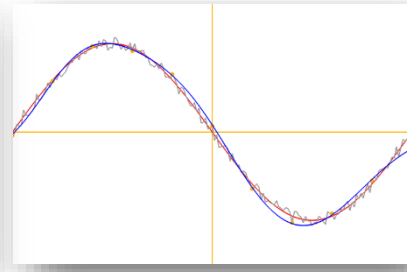
High Noise



Medium Noise



Low Noise



$$y = f(x) + aN$$

$$f(x) = \sin(x)$$

Noise / function distortion levels

$$\text{High Noise} = 0.2 \quad \text{Medium Noise} = 0.1 \quad \text{Low Noise} = 0.05$$

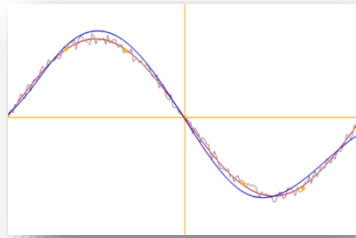
Faster Training Times

What was immediately apparent was how much faster the radial basis function trained not just with low amounts of noise but with high amounts as well. This is not just because there are 1 layer of weights where the multi layered perceptron has 2 (1 set of weights for the hidden layer another for the input layer). But most importantly the radial basis function doesn't have to back propagate error values and adjust weights based on their contribution to that error but rather just multiply each based on the error received. This is much faster than back propagation and is less computationally expensive. This is also far more evident in the project where we can see the training times. The table below shows the training time for the networks. Variables were fixed the only difference being the magnitude of the noise the values for which you can see above.

High Noise	Medium Noise	Low Noise
MLP : 151ms	MLP : 75ms	MLP : 15ms
RBF : 2ms	RBF : 1ms	RBF : 0.4ms

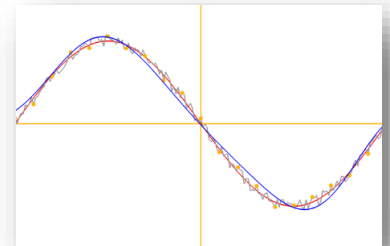
Interpolation & Extrapolation

In testing I found a scenario which would affect the performance. To approximate the function the training set does not need to be sufficiently large to make the radial basis function approximate accurately for the continuous function we are approximating. In the examples you see we create a training set of 20 sample points. However if we set a low number of samples points let's say 6. You



can see the following result. Accuracy is generally maintained even with a medium amount of noise there isn't a massive difference. What's more as there are less training samples we can now have less radial basis functions which results in a quicker training time due to fewer weight to adjust and fewer samples to converge to. The radial basis function is far better at approximating a function where there are a small number of samples are provided at least compared to a multi layered perceptron. However the radial

basis function can only approximate the function when it is given data samples that are within the area it approximates. If it is not given data points that is evenly spread out across the approximation area or if we try to approximate areas of the function where training data isn't given the radial basis function will not be able to deal with it unlike the multilayered perceptron which can as only local calculation is possible.



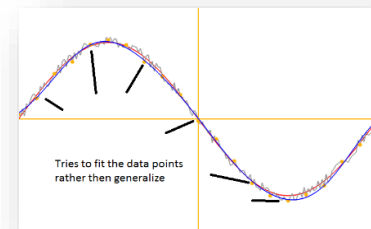
The number of radial basis functions

When dealing with the number of radial basis functions we can split down the situations into 3 separate situations. One where we use too many radial basis functions. Another when we use very few and finally one where we get the balance just about right.

High number of radial basis functions

$$\text{Number of radial basis functions} = 20$$

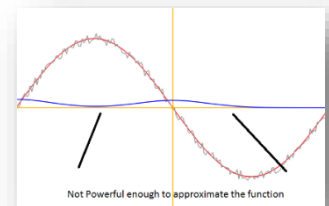
During tests after increasing the number of radial basis functions I begin to see the network not generalize but instead map to the noisy data samples rather than approximate the underlying function. This also results in a longer training time even though this will not result in the training time being worse than the multi layered perceptron it certainly gets worse when you add additional radial basis functions that aren't needed.



Low number of radial basis functions

$$\text{Number of radial basis functions} = 2$$

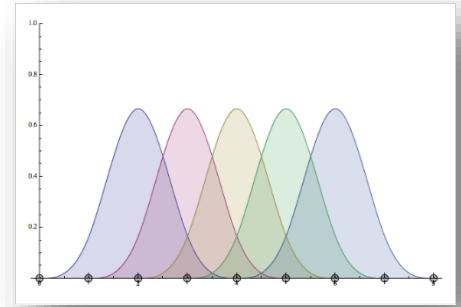
During my tests I found that the radial basis function network is actually very good at approximating from a small number of radial basis function. At least a lot better than the multi layered perceptron and again this leads to a shorter training time. However when the number of radial basis functions is very small it is not powerful enough and cannot approximate the data it is given as shown in the image to the right.



The Basis Function Width

Large Width / Large Neighborhood

In my tests I noted that having a large basis function width resulted in a phenomenon known as output smoothing meaning that the function will do more interpolation often becoming less accurate as a result and won't properly generalize the function. In a sense in my tests it overly generalizes resulting in an inaccurate approximation.



Small Width / Small Neighborhood

Having a smaller width which is a scenario that can be easily forced into a neural network by adding a lot of radial basis functions and you get a similar problem as when you have too many radial basis functions. The network won't generalize and will instead try to fit the noisy points which isn't what we want to achieve here. Of course this is just making the network more accurate with the data samples it is given, however we want to generalize not perfectly fit the training data given and thus this isn't a situation that we want as we want to approximate the data.

Positions of the radial basis function center

Evenly spread out centers across input data

I found it important in my tests to evenly spread out the centers of the radial basis functions when you don't map it so the number of centers equals the number of training patterns. In my tests when I spread the centers unevenly I ended up with a graph where some areas of approximation were perfect and others were way off. Either being too accurate and trying to model the noisy data or not being able to generalize a certain point of the function. Evenly spreading out the centers is crucial to consistent approximation. When centers were evenly spread out it resulted in a more consistent approximation and the distance between each centers would have been the same resulting in a more consistent output or result.

Making the number of functions smaller than the size of the training samples

Additionally for this task where we are trying to approximate the function which has been distorted by noise the radial basis function must have less radial basis functions than training data. The reason being is we want the network to be susceptible to a level of error tolerance and interpolate larger areas which is needed to approximate the function underneath.

OVERALL CONCLUSIONS

My overall conclusion from all things considered is that the radial basis function is better for function approximation when the designs are as the ones I have listed in this document. Not only does the radial basis function train quicker as it doesn't have to use the back propagation algorithm but it scales better when given a large amount of noise unlike the multilayered perceptron which seems to become slower the more noise it has to handle. Also radial basis functions are better in the area when they are presented with little training data given that the data is spread out across the function we want to approximate. The shortcoming of the radial basis function is its inability to approximate outside the range of the data it is given. However I still consider the radial basis function design in the document to be far better than the multi layered perceptron due to quicker training times and its lesser computational cost.



Key points

Radial Basis function

- ❖ Far quicker training times compared with the multilayered perceptron
- ❖ No back propagation making the network better at dealing with larger amounts of noise
- ❖ Better at approximating small amounts of training data but is limited to functions which have a full training set as they can't approximate in areas where no training data is given. Only local calculation is possible not global calculation.

Multi Layered Perception

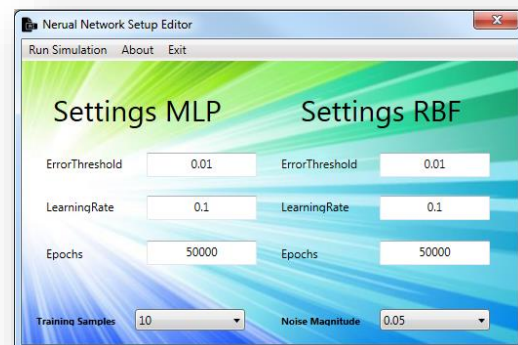
- ❖ Back Propagation results in slower training times
- ❖ Requires a larger training set to approximate the function
- ❖ Can better approximate areas where training data isn't provided as the MLP is capable of global calculation rather than just local calculation.
- ❖ Not guaranteed to converge at all if parameters are setup wrong

USER GUIDE FOR DEMO

The program that you have seen throughout the report can be run on any window PC should you wish to see the program running in action. I have also listed some user instructions below so you can easily navigate the demo yourself.

Guide for the Trainer

Using the trainer written in C#, allows you to set the values for the simulator and should be rather easy to use and don't worry about entering them wrong as I check the inputs before I start the simulator. When you have entered the values you want just press the run simulation menu item and you should be good to go and will show the in program in the image below this.



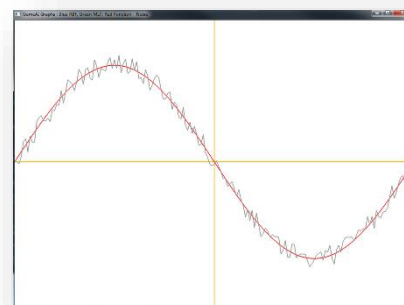
Guide for the Simulator

The simulator written in C++ will take the values outputted by the trainer and try to train the network. Once trained it will bring up a graph like shown below where you can make observations when it comes to the results. Each color of the line corresponds to a specific item which can be seen in the window title. The graph is also intractable by pressing the following keys on your keyboard.

Arrow Right, Moves the graph right, Arrow Down, Moves the graph down, Arrow Left, Moves the graph left, Arrow Up, Moves the graph up

W – Zooms in on the graph S – Zooms out of the graph

F – Toggles full screen mode



LINKS & REFERENCES

Down below I have all the links I used and read over when it came to assembling the following report.

Quotes in the document: <http://www.goodreads.com/quotes/tag/artificial-intelligence>

As I said previously the links down below are articles read in order to generate the report. Please note that work has not been copied these are just papers I used to acquire the knowledge needed to write my project and all of my tests.

Articles Read

<http://neuralnetworksanddeeplearning.com/chap4.html>

http://en.wikipedia.org/wiki/Universal_approximation_theorem

<http://www.naun.org/multimedia/UPress/saed/saed-23.pdf>

<http://web.calstatela.edu/faculty/hmhaska/postscript/neursyst.pdf>

<http://infoscience.epfl.ch/record/82307/files/95-04.pdf>

<http://moodle.uws.ac.uk/mod/resource/view.php?id=199198>

<http://moodle.uws.ac.uk/mod/resource/view.php?id=199199>

