# Feedforward Neural Networks as Radial Basis Function Networks

Edward Venator

EECS 484

## Introduction

In this assignment, a mapping was created from articulated arm joint angles to arm position in the x dimension. This was accomplished using a radial basis function network.

Radial basis function networks allow network responses to be constructed out of weighted sums of arbitrary functions. Although an arbitrary activation function is not biologically plausible, some activation functions can be constructed of several layers of neurons with biologically realistic sigmoid activation functions. In this assignment, Gaussian activation functions were approximated using two layers of neurons with sigmoid activation functions. The first layer, called the alpha layer, consists of perceptrons with hyperbolic tangent activation functions. The hyperbolic tangent function is a sigmoid function with a range of -1 to 1. The second layer, called the beta layer, sums the outputs of the alpha layer using weights of -1 and 1. These weights are selected so that the each beta node has a maximum at one point that slopes off in all directions. By appropriately biasing these nodes, the output closely resembles a Gaussian radial basis function. Once the beta nodes are trained, the third layer is a single node with linear activation function, just as in a normal radial basis function network. The weights of this output (gamma layer) node can be set algebraically with the pseudoinverse, or biologically by random perturbations. In this assignment, the algebraic method was used while tuning the alpha and beta layers, and then a biologically plausible algorithm was created.

## Alpha Layer Bias Selection

Professor Newman’s notes on radial basis function networks describe a method for biasing the alpha nodes of a network of this type. However, his notes make the assumption that the inputs to the system are distributed across the range -1 to 1. Since these inputs do not have that distribution, I rescaled them using Equation 1 below. This rescaling could be incorporated into the weights of the alpha nodes, but for simplicity’s sake, I just rescale the inputs at the beginning of my Matlab script.

|  |  |
| --- | --- |
|  | 1 |

Once the inputs are rescaled, the alpha node biases are generated as described in Poressor newman’s notes. The weights from the inputs are chosen as to be random numbers evenly distributed from -1 to 1. Each bias weight is then chosen to be a random number evenly distributed between where i is an index corresponding to an input (in this case i counts from 1 to 2) and w is the weight from that input into the alpha node. Figure 1 shows surface plot outputs of several alpha nodes, with scaled inputs superimposed in blue. Blue areas correspond to areas rejected by the perceptron (-1), and dark red corresponds to areas accepted by the perceptron (1).

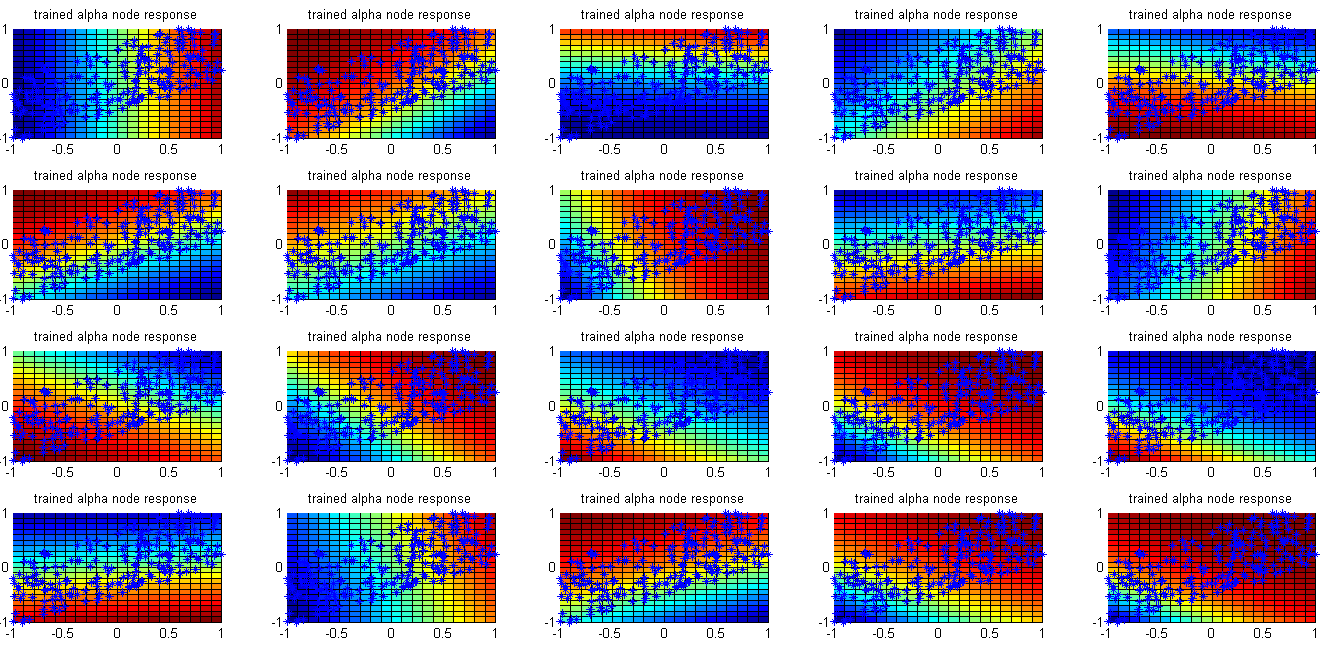


Figure 1

One important parameter to tune in this case is the number of alpha nodes. I initially used a few thousand alpha nodes, but with some tuning determined that as few as 100 alpha nodes could still produce good looking radial basis functions in a properly-tuned beta layer. Fewer alpha nodes of course makes the network faster to simulate, and would make a physical (or biological) implementation of the network more compact.

## Beta Layer Bias Selection

Biasing the beta layer proved trickier than biasing the alpha layer. My method was similar to that described in Professor Newman’s notes. Each beta node is “imprinted” on a randomly chosen input stimulus. No input stimulus is imprinted more than once, and there are fewer beta nodes than input stimuli, which is important to prevent overtraining.

To imprint a beta node, the outputs of the alpha nodes are simulated for the inputs to be imprinted. All alpha nodes with a negative output get a negative weight and all alpha nodes with a positive output get a positive weight. The result is that the summed input into the beta node has a maximum for the imprinted inputs, falling off in every direction.

Since the beta nodes have a logistic (logarithmic sigmoid) activation function, they must be biased to that the input crosses zero at some reasonable radius from the center of the beta node. This bias is dependent on the number of alpha nodes. For alpha nodes with a signum activation function, the beta node bias should be -nalpha + ε, where ε is a small positive number, such as .5 or 1. However, since the hyperbolic tangent function does not always saturate, ε must be larger, and seems to be proportional to nalpha. For 100 alpha nodes, 10 seems to be an appropriate number for ε.

The beta nodes, now tuned, do not yet resemble Gaussians. Because the hyperbolic tangent functions are so steep, the edges of the beta node output are now very steep as well. Figure 2 (below) shows the beta (left) and output (right) responses of the network. The steep responses of the beta nodes result in a rather “spiky” output, and this network has a rather high error of .076 (the weights into the gamma node in this network were trained by the pseudoinverse method, and are therefore optimal).

|  |  |
| --- | --- |
|  |  |

Figure 2

In order to smooth out the beta node responses, I multiplied all of the input weights to the beta layer by a gain term. A gain less than one softens the response of the nodes. I found that a gain of about .05 resulted minimal error for a network with 25 beta nodes. Figure 3 shows that with these smoother beta responses, the output is also smoother and the output error goes down to about .022 (again, using optimal gamma weights).

|  |  |
| --- | --- |
|  |  |

Figure 3

Of course, increasing the number of beta nodes also improves the quality of the fit, and reduces the need for the gain term in the beta weights. With 50 beta nodes, the best fit is achieved with a gain of .125, and the rms error drops to about .013. Figure 4 shows the responses of these beta nodes and the output response. Note the steep drop-off at the edges of the surface.

|  |  |
| --- | --- |
|  |  |

Figure 4

## Gamma Node Bias Selection by Random Perturbations

As discussed above, the pseudoinverse method of tuning the gamma layer is not biologically plausible. An alternative (and biologically plausible) method of tuning is to initialize the gamma layer weights to some number (in this assignment 0) and then perform batch training. On each iteration, the weight vector Wgb is perturbed by adding to it a vector of random numbers, ΔWgb. This ΔWgb vector is generated using a uniform random distribution bounded by a constant ± Δmax. The perturbed weights are used to simulate the entire set of inputs. If the rms error of the output decreases since the last iteration, the perturbed weights become the new weights. If the rms error increases, the perturbation is rejected. As the number of iterations approaches infinity, the rms error converges on the error of the optimal (pseudo inverse) solution, and all perturbations are rejected.

The parameter to be tuned in this training method is the bound of the random distribution Δmax. Choosing a small Δmax causes the network to approach the optimal solution slowly, but a larger Δmax will causes more of the perturbations to be rejected, ultimately resulting in slower training. The ideal Δmax is determined by the scale of the output relative to the outputs of the beta layer. In order to test different values of Δmax, the script was run 10,000 iterations with several values. Figure 5 shows the results. Although larger values initially decline in RMS error much faster, they ultimately reach steady state at a higher level of error.

Figure 5

These tests show that the best value of Δmax is about .5. Figure 6 shows the resulting surface from 10,000 iterations with Δmax=.5 (left) and the optimal surface (right). It is worth noting that the optimal solution has an rms error of .016608, whereas the trained solution has an rms error of .04101.

|  |  |
| --- | --- |
| final trained solution.png | final optimal solution.png |

Figure 6