23 Case Study 1: Function Approximation

Objectives

This chapter represents the first of a series of case studies with neural networks. Neural networks can be used for a wide variety of applications, and it would be impossible to provide case studies for each application. We will limit our presentations to five important application areas: function approximation (aka, nonlinear regression), density function estimation, pattern recognition (aka, pattern classification), clustering and prediction (aka, time series analysis, system identification, or dynamic modeling). For each case study, we will step through the neural network design/training process.

In this chapter, we present a function approximation problem. For function approximation problems, the training set consists of a set of dependent variables (response variables) and one or more independent variables (explanatory variables). The neural network learns to create a mapping between the explanatory variables and the response variables. In the case study we consider in this chapter, the system in question is a smart sensor. A smart sensor consists of one or more standard sensors that are coupled with a neural network to produce a calibrated measurement of a single parameter. In this chapter, we will consider a smart position sensor, which uses the voltages coming from two solar cells to produce an estimate of the location of an object in one dimension.

Theory and Examples

This chapter presents a case study in using neural networks for function approximation. Function approximation consists of defining a mapping between a set of input variables and a corresponding set of output variables. For example, we might want to estimate the price of a home, based on characteristics of the neighborhood, such as tax rate, pupil/teacher ratio in local schools and crime rate. Another example would be estimating octane number of a gasoline product at an oil refinery, based on measurements of reactor temperatures and pressures [FoGi07]. In the case study presented in this chapter, we will consider a smart position sensor system.

Description of the Smart Sensor System

Figure 23.1 illustrates the sensor arrangements for this case study. An object is suspended between a light source and two solar cells. The object casts a shadow on the solar cells, which causes the voltage out of the solar cells to decrease.

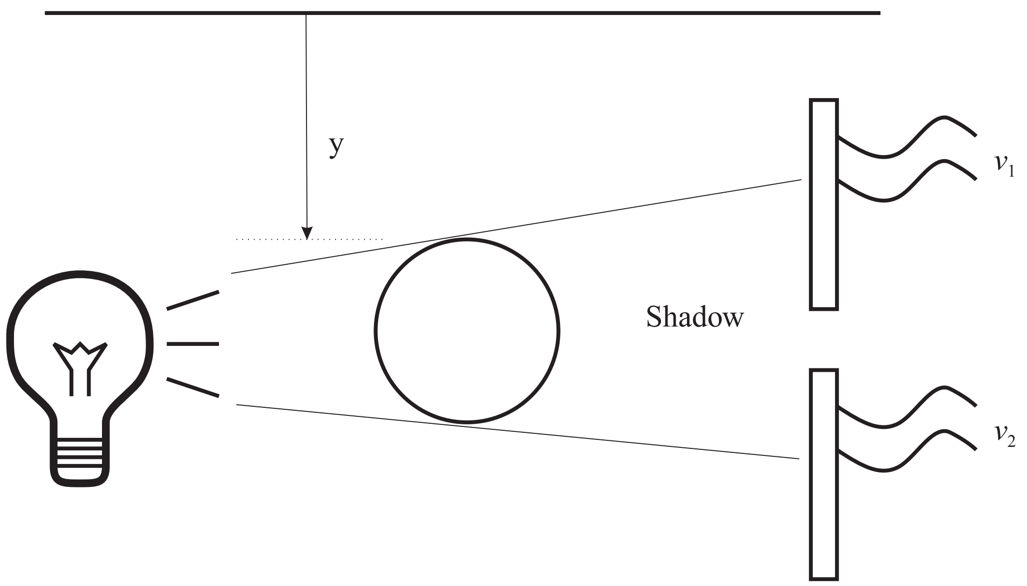

Figure 23.1 Position Sensor Arrangement

As the object position y increases, first the voltage v_1 decreases, then the voltage v_2 decreases, then v_1 increases, and finally v_2 increases. This is demonstrated in Figure 23.2. Our objective is to determine the object position from measurements of the two voltages. Clearly this is a very nonlinear relationship, so a multilayer network will be needed to learn the mapping. This is a classic type of function approximation problem, in which we are trying to learn the inverse of a function. The forward function is the mapping from y to v_1 and v_2. We want to learn the mapping from v_1 and v_2 to y.

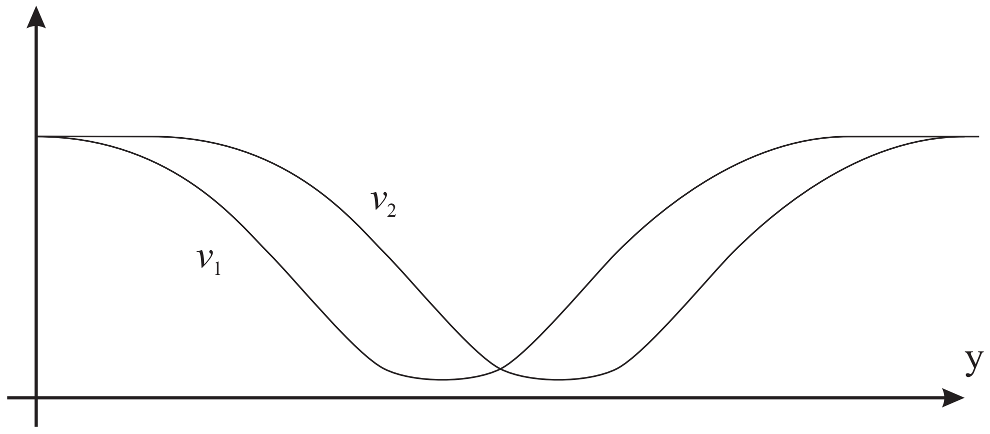

Figure 23.2 Example Solar Cell Outputs vs. Object Position

Data Collection and Preprocessing

In order to collect data for this process, we took measurements of the two solar cell voltages at a number of calibrated positions of the object. The object we used for these experiments was a table tennis ball. The data is displayed in Figure 23.3. There are a total of 67 sets of measurements. Each circle represents a voltage measurement at a calibrated position. The units of position are inches, and the units of voltage are volts. The flat regions at 0 volts for each curve occur where the shadow of the ball completely covers a sensor. If the shadow were large enough to cover both cells at the same time, we would not be able to recover ball position from cell voltages.

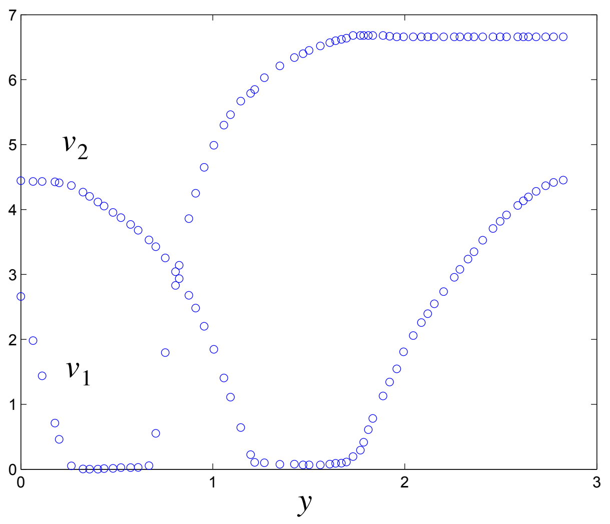

Figure 23.3 Data Collected from Solar Cells

The next step is to divide the data into training, validation and test sets. In this case, because we will be using the Bayesian regularization training technique, we do not need to have a validation set. We did set aside 15% of the data for testing purposes. To perform the division, we arranged the data in order, according to object position, and then selected every sixth or seventh point for testing. This resulted in 10 testing points. The testing points are not used in any way for training the network, but after the network has been completely trained, we will use the testing data as an indicator of future network performance.

The input vector for network training will consist of the solar cell voltages

$$\mathbf{p} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}, \tag{23.1}$$

and the target will be ball position

$$t = y. \tag{23.2}$$

The data were scaled using Eq. (22.1), so that both the inputs and the targets were in the range [-1,1]. The resulting scaled data is shown in Figure 23.4

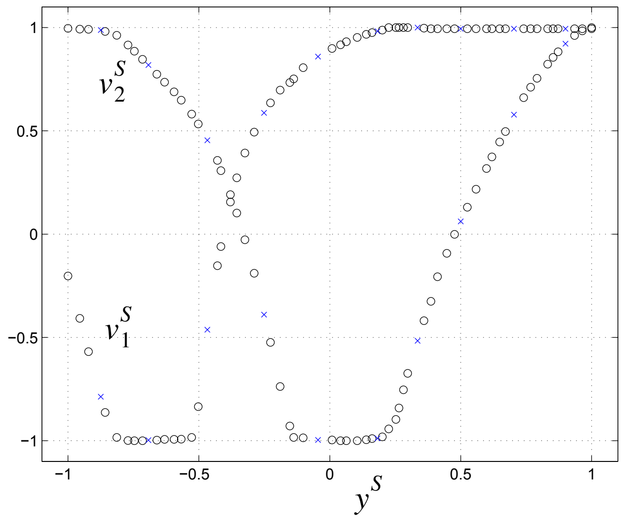

Figure 23.4 Scaled Data

Selecting the Architecture

Because the mapping between the solar cell voltages and the ball position is highly nonlinear, we will use a multilayer network architecture to learn the mapping. We know that there will be two elements in the input vector, which is defined in Eq. (23.1). The single target for the network is the ball position, given in Eq. (23.2).

Figure 23.5 shows the network architecture. We are using the tan-sigmoid transfer function in the hidden layer, and a linear output layer. This is the standard network for function approximation. As we discussed in Chapter 11, this network has been shown to be a universal approximator. There are cases in which two hidden layers are used, but we normally try first with one hidden layer. The number of neurons in the hidden layer, S^1, will depend on the function to be approximated. This is something that cannot generally be known before training. We will have more to say about this in the next section.

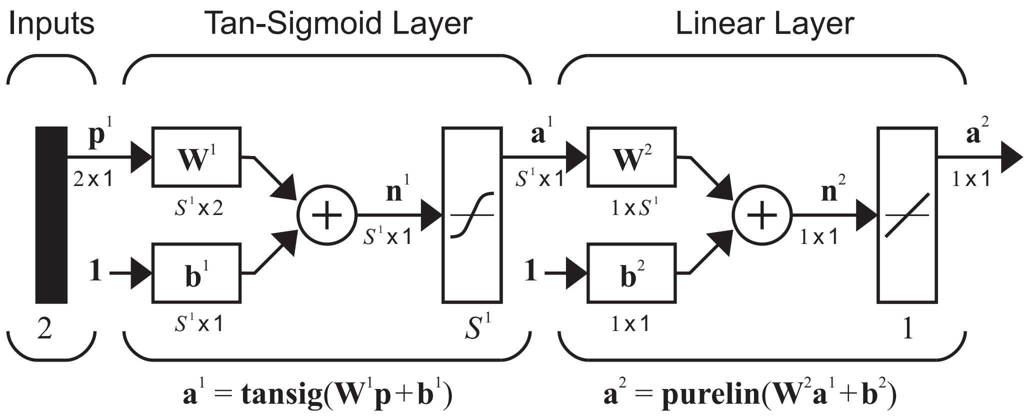

Figure 23.5 Network Architecture

Training the Network

Before beginning the training, we initialized the network weights using the method of Widrow and Nguyen described in Chapter 22. Then we used Bayesian regularization to train the network. Bayesian regularization, which we discussed in Chapter 13, is a very effective algorithm for training multilayer networks to perform function approximation. This algorithm is designed to train networks so that they generalize well, without the need for a validation set. Because the validation set can then be added to the training set, the performance is often better than that obtained with early stopping. (In the next chapter, we will give an example of using early stopping, with a validation set.)

Figure 23.6 illustrates the sum square error versus iteration number, while using the Bayesian regularization training algorithm. We used a network with 10 neurons in the hidden layer ($S^1 = 10$) for this case. The network was trained for 100 iterations, at which time the performance was changing very little.

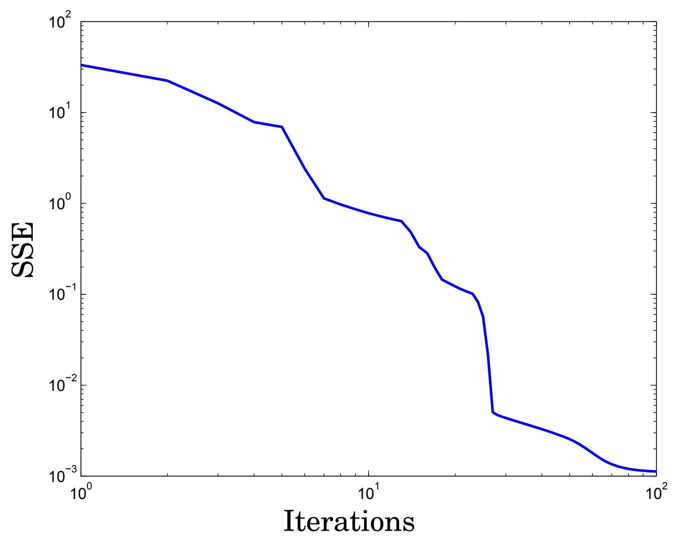

Figure 23.6 Sum Squared Error vs. Iteration Number ($S^1 = 10$)

Training has converged after 100 iterations, but we want to ensure that we have not fallen into a local minimum. For this reason, we want to retrain the network several times, using different initial weights and biases. (We use the Nguyen-Widrow initialization method described in Chapter 22.) Table 23.1. shows the final validation SSE for each of five different training runs. We can see that all of the errors are similar, although the errors are slightly smaller for runs 2, 4 and 5. Any of the weights from these five cases would produce a satisfactory network. We will discuss this in more detail in the next section.

1.121e-003	8.313e-004	1.068e-003	8.672e-004	8.271e-004

Table 23.1. Final Training SSE for Five Different Initial Conditions

Recall from Chapter 13 that the Bayesian regularization algorithm computes a parameter γ, which indicates the effective number of parameters that are being used by the network. In Figure 23.7, we can see the variation of γ during training. It eventually converges to 17.4. There are a total of 41 parameters in this 2-10-1 network, so we are only using about 40% of the weights and biases. For each of the five training runs discussed above, γ converged to values between 17 and 20. This indicates that we might be able to use a smaller network, if we are concerned about the amount of computation required to compute a network response.

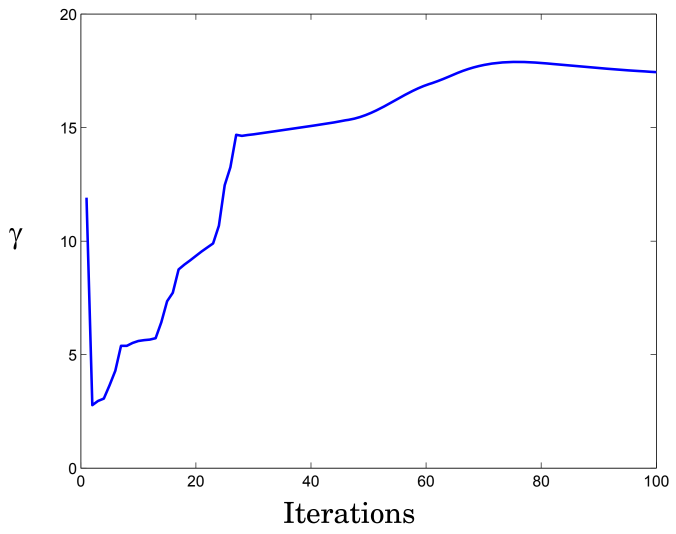

Figure 23.7 Effective Number of Parameters ($S^1 = 10$)

To see whether or not a smaller network would be satisfactory, we trained several networks with different numbers of hidden neurons. Since the effective number of parameters is near 20, we would expect that a network with five hidden neurons (21 weights and biases) might provide an adequate fit. Our experiments produced the results shown in Table 23.2. We can see that the performances of all five networks are roughly equivalent, except for the case where $S^1 = 3$, where the total number of parameters in the network is only 13.

$S^1 = 3$	$S^1 = 5$	$S^1 = 8$	($S^1 = 10$)	$S^1 = 20$
4.406e-003	9.227e-004	8.088e-004	8.672e-004	8.096e-004

Table 23.2. Final Training SSE for Five Different Hidden Layer Sizes

The Bayesian regularization method allows us to train a network of almost arbitrary size, and yet insure that only the required number of parameters is effectively used. If we were concerned about the amount of time required to compute the network output (e.g., for real-time applications), then we would want to use the network with $S^1 = 5$. Otherwise, the original network with $S^1 = 10$ is satisfactory. We don't need to spend a lot of time finding the optimal number of neurons. The training algorithm will insure that we do not overfit.

Validation

An important tool for network validation is a scatter plot of network outputs versus targets, as shown in Figure 23.8 (in normalized units). We expect that for a well trained network the points in the scatter plot will fall close to the 45° output=target line. In this case, the fit is excellent. The figure on the left shows the training data, while the figure on the right shows the testing data. Because the testing data fit is as good as the training data fit, we can be confident that the network did not overfit.

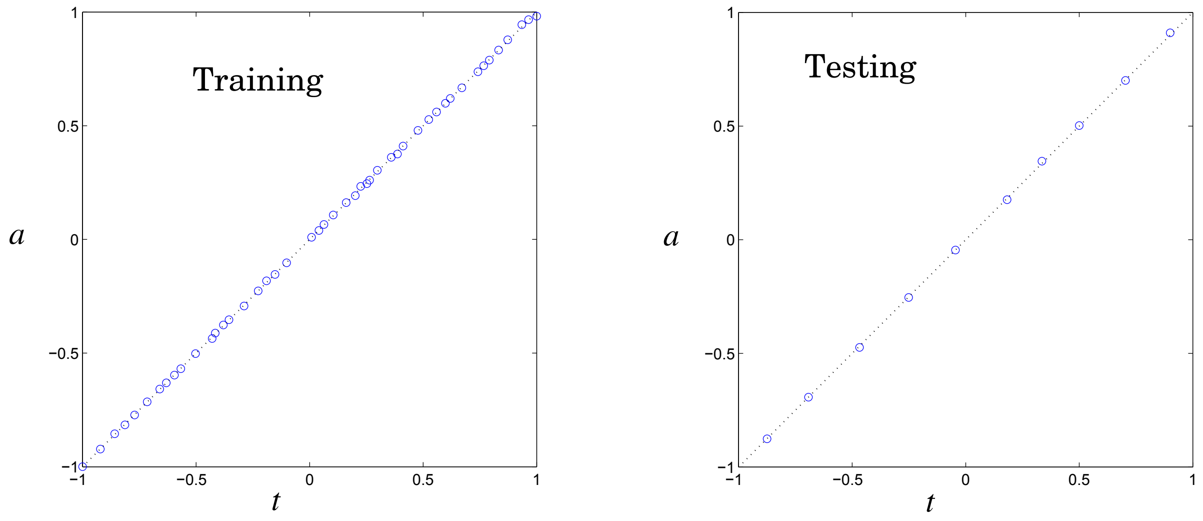

Figure 23.8 Scatter Plots of Network Outputs vs. Targets - Training and Testing Sets

Another useful plot is a histogram of network error, as shown in Figure 23.9. This gives us an idea of the accuracy of the network. For this histogram, we have converted the network output back into units of inches. This is done by applying the reverse of the target preprocessing function to the network output. The reverse of Eq. (22.1), for the targets, is given by

$$\mathbf{a} = (\mathbf{a}^n + 1) .* \frac{(\mathbf{t}^{max} - \mathbf{t}^{min})}{2} + \mathbf{t}^{min} \tag{23.3}$$

where $\mathbf{a}^n$ is the original network output, which was trained to match the normalized target, and $.*$ represents an element-by-element multiplication of two vectors. After the postprocessing operation of Eq. (23.3), the resulting un-normalized output is subtracted from the raw targets, to produce an error in inches. Figure 23.9 shows the distribution of these errors for both training and testing sets. We can see that almost all errors are within one hundredth of an inch. This is within the accuracy of the original measurements, so we cannot expect to do better.

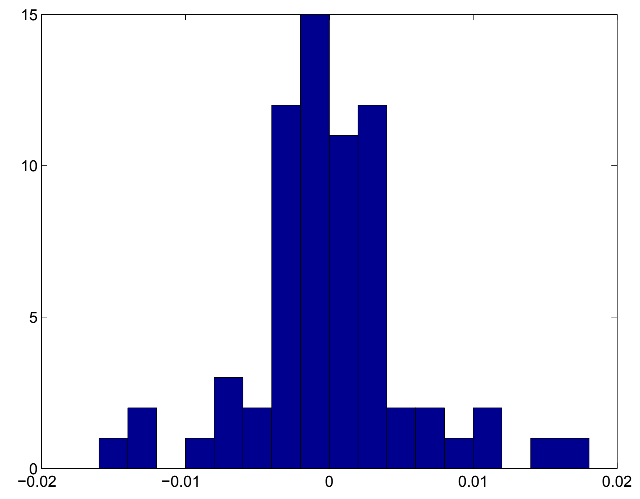

Figure 23.9 Histogram of Position Errors (in Inches)

Because this network has only two inputs, we can plot the trained network response, which is shown in Figure 23.10. (This figure shows the response from original unscaled inputs, in volts, to original unscaled output, in inches.) The blue circles indicate the path that is taken by the voltages as the ball is moved. Notice that the Bayesian regularization training has produced a smooth network response, even though the response is highly nonlinear.

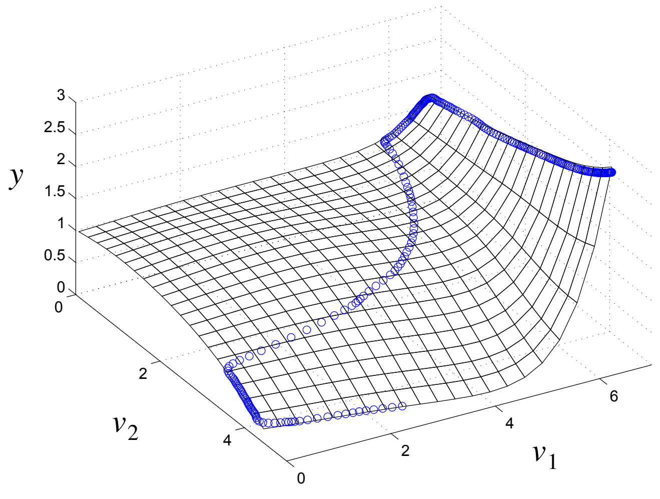

Figure 23.10 Network Response (Original Units)

Another thing to notice about Figure 23.10 is that training data only falls along the blue circles. The form of the network response in other regions is of no importance for the operation of the smart sensor system, since the network will never be used there. If the network were retrained, the shape of the response away from the blue circles might be very different, even though the response near the blue circles will always be the same. This concept is very important for many neural network applications. Often, during normal network operation, only a small portion of the input space will be accessed. The network only has to fit the underlying function in these regions where the network will be used. This means that the size of the data set can be modest, even when the input dimension is large. Of course, in these cases, it is critical that the training data span the full range of potential network operation.

Data Sets

There are two data files associated with this case study:

- ball_p.txt — contains the input vectors in the original data set
- ball_t.txt — contains the target vectors in the original data set

They can be found with the demonstration software, which is described in Appendix C.

Epilogue

This chapter has demonstrated the use of multilayer neural networks for function approximation. This case study is representative of a large class of neural network applications that could be termed “soft sensors” or “smart sensors.” The idea is to use a neural network to fuse several raw sensor outputs into a calibrated measurement of some key variable of interest.

A multilayer network with sigmoid transfer functions in the hidden layers and linear transfer functions in the output layer is well suited to this type of application, and Bayesian regularization is an excellent training algorithm to use in this situation.

In the next chapter, we will look at another neural network application — probability estimation. We will also use multilayer neural networks for that application, but we will change the transfer function in the output layer.

Further Reading

[FoGi07] L. Fortuna, P. Giannone, S. Graziani, M. G. Xibilia, “Virtual Instruments Based on Stacked Neural Networks to Improve Product Quality Monitoring in a Refinery,” *IEEE Transactions on Instrumentation and Measurement*, vol. 56, no. 1, pp. 95–101, 2007.

This paper describes the use of neural networks as soft sensors in a refinery. Measurements of reactor temperatures and pressures are used to predict octane number in a gasoline product.