CHAPTER 1

DATA FITTING WITH LINEAR MODELS

The goal of this chapter is to introduce the following concepts:

- Data fitting and the derivation of the best linear (regression) model
- Iterative solution of the regression model
- Steepest descent methods
- The least mean square (LMS) estimator for the gradient
- The trade-off between speed of adaptation and solution accuracy
- Examples using NeuroSolutions

1.1 INTRODUCTION

The study of neural and adaptive systems is a unique and growing interdisciplinary field that considers adaptive, distributed, and mostly nonlinear systems—three of the ingredients found in biology. We believe that neural and adaptive systems should be considered another tool in the scientist's and engineer's toolbox. They will effectively complement present engineering design principles and help build the preprocessors to interface with the real world and ensure the optimality needed in complex systems. When applied correctly, a neural or adaptive system may considerably outperform other methods.

Neural and adaptive systems are used in many important engineering applications, such as signal enhancement, noise cancellation, classification of input patterns, system identification, prediction, and control. They are used in many commercial products, such as modems, image-processing and -recognition systems, speech recognition, front-end signal processors, and biomedical instrumentation. We expect that the list will grow exponentially in the near future.

The leading characteristic of neural and adaptive systems is their adaptivity, which brings a totally new system design style (Figure 1-1). Instead of being built a priori from specification, neural and adaptive systems use external data to automatically set their parameters. This means that neural systems are parametric. It also means that they are made "aware" of their output through a performance feedback loop that includes a cost function. The performance feedback is utilized directly to change the parameters through systematic procedures called learning or training rules, so that the system output improves with respect to the desired goal (i.e., the error decreases through training).

The system designer has to specify just a few crucial steps in the overall process: He or she has to decide the system topology, choose a performance criterion, and design the adaptive algorithms. In neural systems the parameters are often modified in a selected set of data called the training set and are fixed during operation. The designer thus has to know how to specify the input and desired response data and when to stop the training phase. In adaptive systems the system parameters are continuously

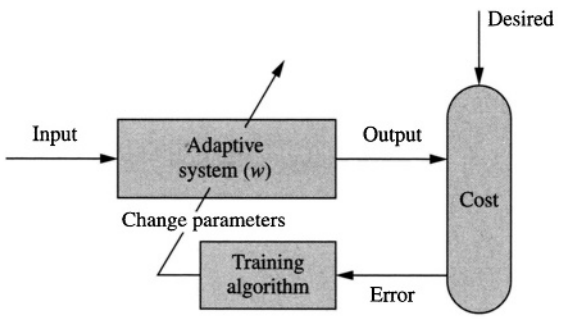


FIGURE 1-1 Adaptive system's design

adapted during operation with the current data. We are at a very exciting stage in neural and adaptive system development because

- We now know some powerful topologies that are able to create universal input-output mappings.
- We also know how to design general adaptive algorithms to extract information from data and adapt the parameters of the mappers.
- We are also starting to understand the prerequisites for generalization, that is, how to guarantee that the performance in the training set extends to the data found during system operation.

Therefore we are in a position to design effective adaptive solutions to moderately difficult real-world problems. Because of the practicality derived from these advances, we believe that the time is right to teach adaptive systems in undergraduate engineering and science curricula.

Throughout this textbook we explain the principles that are necessary to make judicious choices about the design options for neural and adaptive systems. The discussion is slanted toward engineering, both in terminology and in perspective. We are very much interested in the engineering model-based approach and in explaining the mathematical principles at work. We center the explanation on concepts from adaptive signal processing, which are rooted in statistics, pattern recognition, and digital signal processing. Moreover, our study is restricted to model building from data.

1.1.1 Engineering Design and Adaptive Systems

Engineering is a discipline that builds physical systems from human dreams, reinventing the physical world around us. In this respect it transcends physics, which has the passive role of explaining the world, and also mathematics, which stops at the edge of physical reality. Engineering design is like a gigantic Lego construction, where each piece is a subsystem grounded in its physical or mathematical principles. The role of the engineer is to develop the blueprint of the dream through specifications and then to look for the pieces that fit the blueprint. Obviously, the pieces cannot be put together at random, since each has its own principles, so it is mandatory that the scientist or the engineer learn the principles attached to each piece and specify the interface. Normally, this study is done using the scientific method. When the system is physical, we use the principles of physics, and when it is software, we use the principles of mathematics.

Development of the Phone System

A good example of engineering design is the telephone system. Long and meticulous research was conducted at Bell Laboratories on human perception of speech. This created

the specification for the required bandwidth and noise level for speech intelligibility. Engineers perfected the microphone that would translate the pressure waves into electrical waves to meet the specification. Then these electrical waves were transmitted through copper wires over long distances to a similar device, still preserving the required specification. For increased functionality the freedom of reaching any other telephone was added to the system, so switching of calls had to be implemented. This created the phone system. Initially, the switching among lines was done by operators. Then a machine was invented that would automatically switch the calls. Operators were still used for special services such as directory assistance, but now that the fundamental engineering aspects are stable, we are asking machines to automatically recognize speech and directly assist callers.

The development of the phone system is an excellent example of engineering design. Once we have a vision, we try to understand the principles at work and create specifications and a system architecture. The fundamental principles at work are found by applying the scientific method. The phenomenon under analysis is first studied with physics or mathematics. The importance of models is that they translate general principles, and through *deduction* we can apply them to particular cases like the ones we are interested in. These disciplines create approximate models of the external world using the principle of divide and conquer. The problem is divided in manageable pieces, each is studied independently of the others, and protocols among the pieces are drawn such that the system can work as a whole, meeting the specifications drawn a priori. This is what engineering design is today.

The scientific method has been highly successful in engineering, but let us evaluate it in broad terms. First, engineering design requires the availability of a model for each subsystem. Second, when the number of pieces increases, the interactions among the subsystems increase exponentially. Fundamental research will continue to provide a steady flux of new physical and mathematical principles (provided the present trend of reduced federal funding for fundamental science is reversed), but the exponential growth of interactions required for larger and more sophisticated systems is harder to control. In fact, at this point in time, we simply do not have a clear vision of how to handle complexity in the long term. But there are two more factors that present major challenges: the autonomous interaction of systems with the environment and the optimality of the design. We will discuss these now.

Humans have traditionally mediated the interaction of engineering systems with the external world. After all, we use technology to reduce our physical constraints, so we have traditionally maintained control of the machines we build. Since the invention of the digital computer, there has been a trend to create machines that interact directly with the external world without a human in the loop. This brings the complexity of the external world directly into engineering design. We are not yet totally prepared for this, because our mathematical and physical theories about the external world are mere approximations—very good approximations in some cases, but rather poor in others.

This disturbs the order of engineering design and creates performance problems (the worst subsystem tends to limit the performance of the full system).

Mars Pathfinder Mission

When machines have to autonomously interact with the environment or operate near the optimal set point, we cannot specify all the functions a priori and in a deterministic way. Take, for instance, the Mars Pathfinder mission. It was totally impossible to specify all the possible conditions that the rover Sojourner would face, even if remotely controlled from Earth, so the problem could not be solved by a sequence of instructions determined a priori in JPL's laboratory. The vehicle was given high-level instructions (way points) and was equipped with cameras and laser sensors that would see the terrain. The information from the sensors was analyzed and catalogued in general classes. For each class a procedure was designed to accomplish the goal of moving from point A to point B. This is the type of engineering system we will build more and more of in the future.

The big difference between the initial machines and Sojourner is that the environment is intrinsically in the loop of the machine function. This brings a very different set of problems, because, as we said earlier, the environment is complex and unpredictable. If our physical model does not capture the essentials of the environment, errors accumulate over time, and the solution becomes impractical. We thus no longer have the luxury of dictating the rules of the game, as we did in the early machine-building era. It turns out that animals and humans do Sojourner-type tasks effortlessly.

System optimality is also a rising concern to save resources and augment the performance/price ratio. We might think that optimally designing each subsystem would bring global optimality, but this is not always true. Optimal design of complex systems is thus a difficult problem that must also take into consideration the particular type of system function, meaning that the complexity of the environment is once again present. We can conclude that the current challenges faced in engineering are the complexity of the systems, the need for optimal performance, and the autonomous interaction with the environment that will require some form of intelligence. These are the challenges for engineering in the 21st century and beyond.

Whenever there is a new challenge, we should consider new solutions. Quite often the difficulty of a task is also linked to the particular method we are using to find the solution. Is building machines by specification the only way to proceed?

Let us look at living creatures from an engineering systems perspective. The cell is the optimal factory, building directly from the environment at the fundamental molecular level what it needs to carry out its function. The animals we observe today interact efficiently with the environment (otherwise they would not have survived); they work very close to optimality in terms of resources (otherwise they would have been replaced in their niche by more efficient animals); and they certainly are complex. Biology has,

in fact, already conquered some of the challenges we face in building engineering systems, so it is worthwhile to investigate the principles at work.

Biology has found a set of *inductive* principles that are particularly well tuned to the interaction with a complex and unpredictable environment. These principles are not known explicitly but are being intensively studied in biology, computational neurosciences, statistics, computer science, and engineering. They involve extraction of information from sensor data (feature extraction), efficient learning from data, creation of invariants and representations, and decision making under uncertainty. In a global sense, autonomous agents have to build and fit models to data through their daily experience; they have to store these models, choose which shall be applied in each circumstance, and assess the likelihood of success for a given task. An implicit optimization principle is at play, since the goal is to do the best with the available information and resources.

From a scientific perspective, biology uses *adaptation* to build optimal system functionality. The anatomical organization of the animal (the wetware) is specified in the long term by the environment (through evolution), and in the short term it is used as a constraint to extract in real time the information that the animal needs to survive. At the nervous-system level, it is well accepted that the interaction with the environment molds the wetware using a learning-from-examples metaphor.

1.1.2 Experimental Model Building

The problem of data fitting is one of the oldest in experimental science. The real world tends to be very complex and unpredictable, and the exact mechanisms that generate the data are often unknown. Moreover, when we collect physical variables, the sensors are not ideal (of finite precision, noisy, with constrained bandwidth, etc.), so the measurements do not represent the real phenomena exactly. One of the quests in science is to estimate the underlying data model.

The importance of inferring a model from the data is to apply mathematical reasoning to the problem. The major advantage of a mathematical model is the ability to understand, explain, predict, and control outcomes in the natural system [Casti, 1989]. Figure 1-2 illustrates the data-modeling process. The most important advantage of the

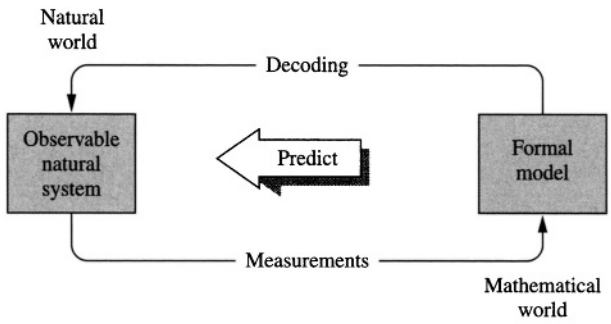


FIGURE 1-2 Natural systems and formal models

existence of a formal equivalent model is the ability to predict the natural system's behavior at a future time and to control its outputs by applying appropriate inputs.

In this chapter we address the issues of fitting data with linear models, which is called the *linear regression* problem [Dunteman, 1989]. Notice that we have not specified what the data is, because it is really immaterial. We are seeking relationships between the values of the external (observable) variables of the natural system in Figure 1-2. This methodology can therefore be applied to meteorological data, biological data, financial data, marketing data, engineering data, and so on.

1.1.3 Data Collection

The data-collection phase must be carefully planned to ensure that

- Data will be sufficient.
- Data will capture the fundamental principles at work.
- Data is as free as possible from observation noise.

Table 1-1 presents a data example with two variables (x, d) in tabular form. the measurement x is assumed error free, and d is contaminated by noise. From table 1-1 very little can be said about the data, except that there is a positive trend between the variables (i.e., when x increases d also increases). Our brain is somehow able to extract much more information from pictures than from numbers, so data should first be plotted before performing data analysis. Plotting the data allows verification, assures the researcher that the data was collected correctly, and provides a "feel" for the relationships that exist in the data (e.g., natural trends).

TABLE 1-1 Regression Data

x	d
1	1.72
2	1.90
3	1.57
4	1.83
5	2.13
6	1.66
7	2.05
8	2.23
9	2.89
10	3.04
11	2.72
12	3.18

1.2 LINEAR MODELS

From the simple observation of Figure 1-3, it is obvious that the relationship between the two variables x and d is complex, if we assume that no noise is present. However, there is an approximately linear trend in the data. The deviation from a straight line could be produced by noise, and underlying the apparent complexity could be a very simple (possibly linear) relationship between x and d, that is

$$d \approx wx + b \tag{1.1}$$

or more specifically,

$$d_i = wx_i + b + \varepsilon_i = y_i + \varepsilon_i \tag{1.2}$$

where ε_i is the instantaneous error that is added to y_i (the linearly fitted value), w is the line slope, and b is the d axis intercept (or bias). Assuming a linear relationship between x and d has the appeal of simplicity. The data-fitting problem can be solved by a linear system with only two free parameters, the slope w and the bias b.

The system of Figure 1-4 is called the linear *processing element* (PE), or *Adaline* (for adaptive linear element), and it is very simple. It is built from two multipliers and one adder. The multiplier w *scales the input,* and the multiplier b *is a bias,* which can also be thought of as an extra input connected to the value $+1$. The parameters (b, w) have different functions in the solution. We will be particularly interested in studying the dependence of the solution on the parameter(s) that multiply the input x_i.

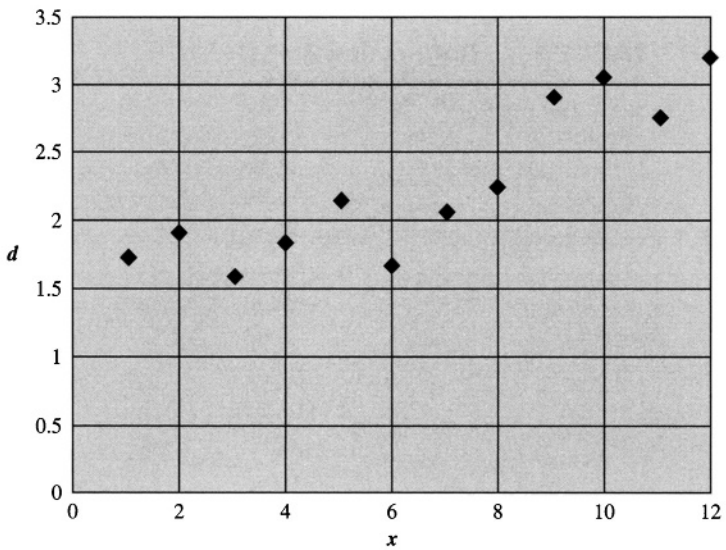


FIGURE 1-3 Plot of x versus d

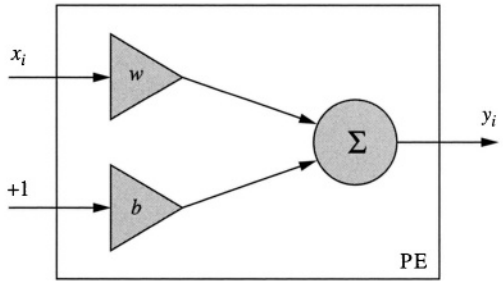


FIGURE 1-4 Linear regressor

NEUROSOLUTIONS EXAMPLE 1.1

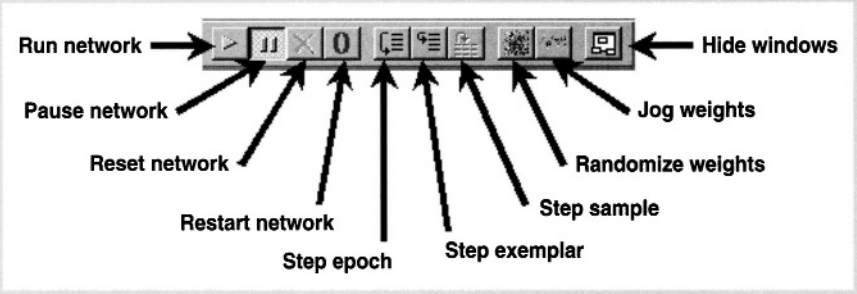
The linear processing element in NeuroSolutions

The goal of this book is to demonstrate as many concepts as possible through demonstrations and simulations. NeuroSolutions is a very powerful neural network/adaptive system design and simulation package that we will use for the demonstrations. We highly recommend that you read through the NeuroSolutions Tutorial in Appendix B. NeuroSolutions constructs adaptive systems in a Lego style, that is, component by component. The components are chosen from palettes, selected with the mouse, and dropped into the large window called the Breadboard. This object-oriented methodology allows for the simple creation of adaptive systems by simply dragging and dropping components, connecting them, and then adjusting their parameters. Particularly in the early chapters we will automatically create the adaptive systems for you through a set of macros. This will shield you from the details of NeuroSolutions until you have a better grasp of the fundamentals of adaptive systems and the use of NeuroSolutions.

In this first example we introduce a few simple components. The first component required in any simulation is an input component, which belongs to the Axon family. Its function is to receive data from the computer file system or from signal generators within the package. In this case, we will add a file input component to the input Axon to read in the data from Table 1-1. The linear PE shown in Figure 1-4 can be constructed with a Synapse and a Bias Axon. The Synapse implements a sum of products and the Bias Axon adds the bias. The output of such a system is exactly Eq. (1.1). The Controller manages the system and controls the firing of data

through the system. Since Table 1-1 has 12 data points, the Controller is configured to send 12 points through the system.

The purpose of this example is to display the output of the linear PE, which is a line, and to modify its location in the space by entering different slope and bias values. To display the input and regression line, we use the Data Storage component (stores 12 samples) and the Scatter Plot component. The Scatter Plot component allows us to plot the input (x axis) versus the system response (y axis). We also add two Matrix Editor boxes to allow you to change the values of the two parameters: the weight (slope) and bias (y intercept). After changing these parameters, you use the Control Palette to run the network.



The *Run* button (a green triangle) tells the Controller to send the data through the network. The other buttons are not important now but will be used and explained later. Run the NeuroSolutions example by clicking on the yellow NeuroSolutions icon available in the CD-ROM version of the text. It will walk you through the creation of the Breadboard and allow you to see how the regression line changes as you change the weight (slope) and bias (y intercept).

1.3 LEAST SQUARES

We face a problem when trying to fit a straight line to the noisy observations of Table 1-1. A single line will fit any two observations (two points define a line), but it is unlikely that all points will fall on exactly the same line. Since no single line will fit every point, a global property of the points is needed to find the best fit. The problem of fitting a line to noisy data can be formulated as follows: What is the best choice of (w, b) such that the fitted line passes the closest to all the points?

Least squares solves the problem of fitting a line to data by finding the line for which the sum of the square deviations (or residuals) in the d direction (the noisy variable direction) are minimized. The fitted points in the line will be denoted by $\tilde{d}_i = b + wx_i$. The residuals are defined as $\varepsilon_i = d_i - \tilde{d}_i$. The fitted points $\tilde{d}_i$ can also be interpreted as approximated values of d_i estimated by a linear model when the input x_i is known:

$$d_i - (b + wx_i) = d_i - \tilde{d}_i = \varepsilon_i \tag{1.3}$$

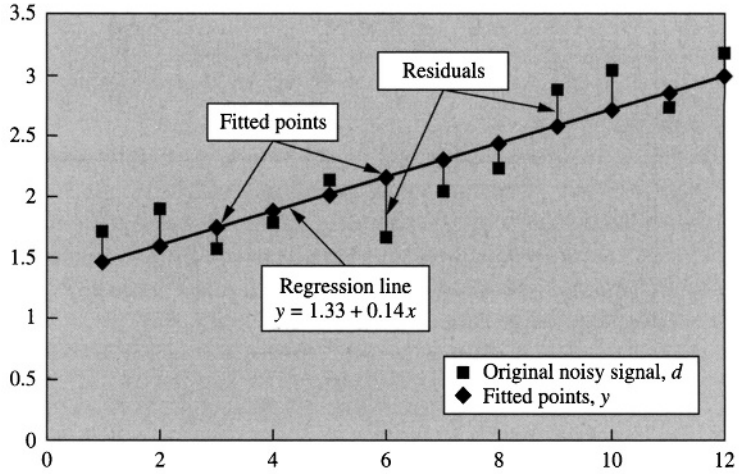


FIGURE 1-5 Regression line showing the deviations

This linear model will be called the *linear regressor.* Estimated quantities will be denoted by the tilde (˜) throughout the book. The outputs of the linear system of Figure 1-4 are the fitted points $\tilde{d}_i = y_i$ in Figure 1-5. To pick the line that best fits the data, we need a criterion to determine which linear estimator is the "best." The average sum of square errors J (also called the *mean square error,* MSE) is a widely utilized performance criterion given by

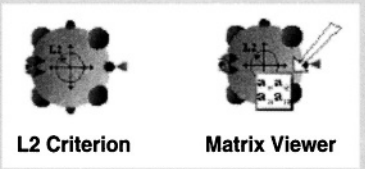
$$J = \frac{1}{2N} \sum_{i=1}^{N} \varepsilon_i^2 \tag{1.4}$$

where N is the number of observations. To simplify the notation, we sometimes drop the top index in the sum.

NEUROSOLUTIONS EXAMPLE 1.2

Computing the MSE for the linear PE

To create a simulation that displays the MSE, we have to add a new component to the Breadboard, the L2 Criterion. The L2 Criterion implements the mean square error of Eq. 1.4. The L2 Criterion requires two inputs to compute the MSE: the system output and the desired response. We will attach the L2 Criterion to the output of the linear PE (system output) and attach a File Input component to the L2 Criterion to load in the value of the desired response from Table 1-1. To visualize the MSE, we will place a Matrix Viewer probe over the L2 Criterion (cost access point). This Matrix Viewer simply displays the data from the component that it resides over—in this case, the mean square error.



Run the demonstration and try to set the slope and bias to minimize the mean square error. Compute the error by hand according to Eq. 1.4 and see whether it matches the value displayed.

Our goal is to minimize J analytically, which according to Gauss[1] can be done by taking its partial derivative with respect to the unknowns and equating the resulting equations to zero:

$$\begin{aligned} \frac{\partial J}{\partial b} &= 0 \\ \frac{\partial J}{\partial w} &= 0 \end{aligned} \tag{1.5}$$

which yields, after some manipulation,

$$b = \frac{\sum_i x_i^2 \sum_i d_i - \sum_i x_i \sum_i x_i d_i}{N[\sum_i (x_i - \bar{x})^2]} \qquad w = \frac{\sum_i (x_i - \bar{x})(d_i - \bar{d})}{\sum_i (x_i - \bar{x})^2} \tag{1.6}$$

where an overbar represents the variable's mean value; for example, $\bar{x} = (1/N)\sum_{i=1}^{N} x_i$.

Least Squares Derivation

In Eq. 1.4 we substitute the value of the error given by Eq. 1.3 and take the derivative expressed by Eq. 1.5 to obtain

$$\begin{cases} \dfrac{\partial J}{\partial b} = \dfrac{1}{2N}\displaystyle\sum_i \frac{\partial (d_i - wx_i - b)^2}{\partial b} = -\sum_i \frac{1}{N}(d_i - wx_i - b) = 0 \\ \dfrac{\partial J}{\partial w} = \dfrac{1}{2N}\displaystyle\sum_i \frac{\partial (d_i - wx_i - b)^2}{\partial w} = \sum_i \frac{1}{N}(d_i - wx_i - b)x_i = 0 \end{cases} \tag{1B.1}$$

[1]Karl Friedrich Gauss (1777–1855) was a mathematical genius who proposed the use of least squares to solve sets of linear equations. He realized that a quadratic equation was obtained in optimization problems involving Gaussian distribution models (after taking the logarithm), which leads to an easy solution for the optimum.

Note that to simplify the notation, we omit the limits of the variable in the sum. We do this throughout the text when no confusion arises. Operating further, we get

$$\begin{aligned} \sum_{i=1}^{N} d_i &= Nb + w\sum_{i=1}^{N} x_i \\ \sum_{i=1}^{N} x_i d_i &= b\sum_{i=1}^{N} x_i + w\sum_{i=1}^{N} x_i^2 \end{aligned} \tag{1B.2}$$

The set of Eq. 1B.2 is called the normal equations. The solution of this set of equations is

$$b = \frac{\sum_i x_i^2 \sum_i d_i - \sum_i x_i \sum_i x_i d_i}{N\sum_i x_i^2 - (\sum_i x_i)^2} \qquad w = \frac{\sum_i x_i d_i - \dfrac{\sum_i x_i \sum_i d_i}{N}}{\sum_i x_i^2 - \dfrac{(\sum_i x_i)^2}{N}} \tag{1B.3}$$

which provides the coefficients for the *regression line of d on x*. The summations run over the input-output data pairs. Equation 1B.2 is solved by computing the value of b from the first equation and substituting it in the second equation to obtain w as a function of x and d. Then the value of w is substituted in the first equation to finally obtain b as a function of x and d (variable elimination). It is easy to prove that the regression line passes through the point

$$\left(\frac{\sum_i x_i}{N}, \frac{\sum_i d_i}{N}\right)$$

which is called the centroid of the observations. The denominator of the slope parameter of w and b is the corrected (for the mean) sum of squares of the input.

This procedure to determine the coefficients of the line is called the least square method. If we apply these equations to the data of Table 1-1, we get the regression equation (best line through the data):

$$y = 0.13951x + 1.33818 \tag{1.7}$$

The least square computation for a large data set is time-consuming, even with a computer.

NEUROSOLUTIONS EXAMPLE 1.3

Finding the minimum error by trial and error

Enter these values for the slope and bias by typing them in the respective Edit boxes. Verify that with these values the error is the smallest. Change the values slightly (in

either direction) and see that the MSE increases. Enter a negative slope and see how the error increases a lot. For the negative slope, what is the value of the bias that gives the smallest error? Note that when one of the coefficients is wrong, the value of the other for best performance is also wrong; that is, they are coupled.

It is important to explore the NeuroSolutions Breadboards. The best way to accomplish this is to open the Inspector associated with each icon. Select a component with the mouse. Then press the right mouse button, and select properties. The Inspector will appear in the screen. The Inspector has fields that allow us to configure the NeuroSolutions components and tell us what settings are being used. For instance, go to the input Axon and open the Inspector. You will see that it has one input, one output, and no weights (go to the Soma level to look at the weights). If you do the same in the Synapse, you will see that it also has a single input and output and one weight, which happens to be our slope parameter. The Bias Axon has a single input, a single output, and a single weight, which is the system's bias.

The large barrel on the input Axon is a probe that collects data. Since the barrel is placed on the Activity point, it is storing the 12 data samples that are injected into the network. This is exactly what gets displayed in the x axis of the Scatter Plot. The y axis values are sent from the L2 Criterion by the small barrel (a Data Transmitter), so the Scatter Plot is effectively displaying the pairs of points (x_i, d_i). Likewise, it is also displaying the output of the system in blue, that is the pairs of points (x_i, y_i).

If you want to know what the component is and what it does, just go to the NeuroSolutions control bar, select the arrow icon with the question mark, and click on the component that you want to know about (this is called context-sensitive help).

1.3.1 Correlation Coefficient

We have found a way to compute the regression equation, but we still do not have a measure of how successfully the regression line represents the relationship between x and d. The size of the mean square error (MSE) can be used to determine which line best fits the data, but it doesn't necessarily reflect whether a line fits the data tightly because the MSE depends on the magnitude of the data samples. For instance, by simply scaling the data, we can change the MSE without changing how well the data is fit by the regression line. The *correlation coefficient* (r) solves this problem. By definition, the correlation coefficient between two random variables x and d is

$$r = \frac{\frac{\sum_i (x_i - \bar{x})(d_i - \bar{d})}{N}}{\sqrt{\frac{\sum_i (d_i - \bar{d})^2}{N}} \sqrt{\frac{\sum_i (x_i - \bar{x})^2}{N}}} \tag{1.8}$$

The numerator is the *covariance* of the two variables (see Section 4.11), and the denominator is the product of the corresponding *standard deviation.*

Variance

Data collected from experiments is normally very complex and difficult to describe by few parameters. The mean and the variance are statistical descriptors of data clusters that are normally utilized in such cases.

The mean of N samples is defined as

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$$

A physical interpretation for the mean is the center of mass of a body made up of samples of the same mass. It is the first moment of the probability density function (pdf).

We can have very different data distributions with the same mean, so the mean is not a powerful descriptor. Another descriptor very often used is the variance, which is defined as

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

The variance is the second moment around the mean, and it measures the dispersion of samples around the mean. The square root of the variance is called the standard deviation. Mean and variance are much better descriptors of data clusters. In fact, *they define* univocally Gaussian distributions, which are very good models for many real-world phenomena.

The correlation coefficient is confined to the range $[-1, 1]$. When $r = 1$, there is a perfect positive linear correlation between x and d; that is, they covary, which means that they vary by the same amount. When $r = -1$, there is a perfectly linear negative correlation between x and d; that is, they vary in opposite ways (when x increases, y decreases by the same amount). When $r = 0$, there is no correlation between x and d; that is, the variables are called *uncorrelated.* Intermediate values describe partial correlations. In our example $r = 0.88$, which means that the fit of the linear model to the data is reasonably good. Notice that the correlation coefficient is a property of the data, as we can see from Eq. 1.8 (it is independent of the model). However, the value r^2 also represents the amount of variance in the data captured by the optimal linear regression.

Correlation Coefficient and Linear Regression

Consider estimating a random variable d by a constant b according to the mean square estimation,

$$\min_b E[(d - b)^2] = E[d^2] - 2bE[d] + b^2$$

where $E[.]$ is the expected value operator (see Appendix A). The best b is obtained by taking the derivative with respect to b and setting the result to zero, yielding $b^* = E[d]$. If we substitute this value into the equation, we obtain the variance of d as the smallest error.

Now if we try to approximate d by $wx + b$ we obtain

$$\min_{w,b} E[(d - wx - b)^2]$$

and the best $b^* = E[d] - wE[x]$. Therefore the best w can be found solving the problem

$$\min_{w} E[((d - E[d]) - w(x - E[x]))^2]$$

It is easy to show by differentiation that the best w^* is

$$w^* = r\frac{\sigma_d}{\sigma_x} \tag{1B.4}$$

Therefore the minimum mean square error linear estimator for d is

$$\hat{d} = w^*x + b^* = r\sigma_d\left(\frac{x - E[x]}{\sigma_x}\right) + E[d] \tag{1B.5}$$

The term in parentheses is a zero-mean unit variance version of the input x, so the product with σ scales it by the variance of d. The term $E[d]$ just guarantees the correct mean. It is interesting to note that if d and x are uncorrelated, the best estimate of d is its mean. However, if x and d are exactly correlated ($r = \pm 1$ in Eq. 1B.5), the best estimate is highly improved. The minimum mean square error is

$$\min_{w} E[((d - E[d]) - w^*(x - E[x]))^2] = \sigma_d^2(1 - r^2)$$

This equation shows that in fact r^2 can be interpreted as the amount of variance in the data that is captured by the linear model.

There is a very interesting interpretation of the mean square estimation solution. Note that

$$-E[(x - E[x])(d - E[d])] + 2w^*\sigma_x^2 = 0 = E[\{(d - E[d]) - w^*(x - E[x])\}(x - E[x])] \tag{1B.6}$$

which means that the error (the quantity inside the curly braces) is orthogonal to the input.

The method of least squares is very powerful. Estimation theory [Melsa and Cohn, 1978] says that the least square estimator is the best linear unbiased estimator (BLUE), since it has no bias and has minimal variance among all possible linear estimators. Least squares can be generalized to higher-order polynomial curves, such as quadratics, cubics, and so on (the *generalized least squares*). In this case, nonlinear regression

models are obtained. More coefficients need to be computed, but the methodology still applies. Regression can also be extended to multiple variables, as we will do later in the chapter. The dependent variable d in multiple variable regression is a function of a vector $\mathbf{x} = [x_1, \ldots, x_D]^T$, where T means the transpose and D is the number of inputs. In this book, vectors are denoted by bold letters. In the multivariate case, the regression line becomes a *hyperplane* in the space $x_1, x_2, \ldots, x_D$.

1.4 ADAPTIVE LINEAR SYSTEMS

1.4.1 Least Squares as a Search for the Parameters of a Linear System

The purpose of least squares is to find parameters (b, w) that minimize the difference between the system output y_i and the desired response d_i. Regression is effectively computing the optimal parameters of an interpolating system (linear in this case) that predicts the value of d from the value of x.

Figure 1-6 shows graphically the operation of adapting the parameters of the linear system. The system output y is always a linear combination of the input x with the bias, so it has to lie on a straight line of equation $y = wx + b$. Changing b modifies the y intercept, while changing w modifies the slope. Therefore we conclude that the goal of linear regression is to adjust the position of the line to minimize the average square difference between the y values (on the line) and the cloud of points d_i (i.e., the criterion J).

The key point is to recognize that the error contains information that can be used to optimally place the line. Figure 1-6 shows this by including a subsystem that accepts the error as input and modifies the parameters of the system. Thus, the error ε_i is fed back to the system and indirectly affects the output through a change in the parameters (b, w). Effectively, the system is made "aware" of its performance through the error. With the incorporation of the mechanism that automatically modifies the system parameters, a very powerful linear system can be built that will constantly seek optimal parameters. Such systems are called neural and adaptive systems and are the focus of this book.

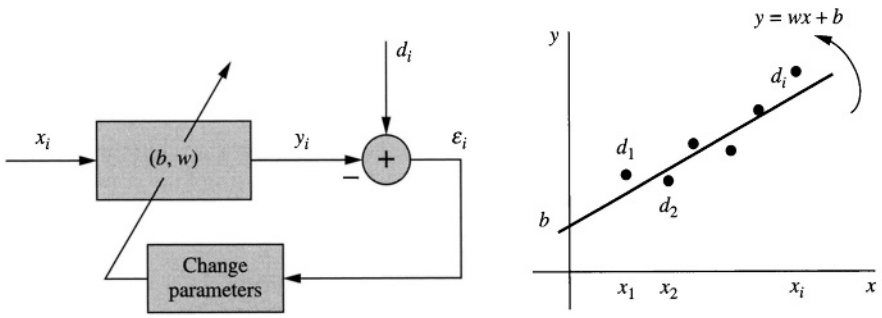


FIGURE 1-6 Regression as a linear system design problem

1.4.2 Neural and Adaptive Systems

Before pursuing the study of adaptive systems, it is important to reflect briefly on the implications of neural and adaptive systems in engineering design. System design usually begins with specifications. First, the problem domain is studied and modeled, specifications are established, and then a system is built to meet the specifications. The key point is that the system is built to meet the current specifications and will always use the designed set of parameters, even if the external conditions change.

Here we are proposing a very different system design approach based on adaptation, which has a biological flavor to it. In the beginning the system parameters may be way off, creating a large error. However, through the feedback from the error, the system can change its parameters to decrease the error as much as possible. The system's "experience" with the data designs the best set of parameters. An adaptive system is more complex because it has to not only accomplish the desired task but also be equipped with a subsystem that adapts its parameters. But notice that even if the data changes in the future, this design methodology will modify the system parameters so that the best possible performance is obtained. Additionally, the same system can be used for multiple problems.

There are basically two ways to adapt the system parameters: *supervised learning* and *unsupervised learning*. The method described until now belongs to supervised learning because there is a desired response. Later on in the book we will find other methods that also adapt the system parameters, but using only an internal rule. Since there is no desired response, these methods are called unsupervised. We will concentrate here on supervised learning methods.

The ingredients of supervised adaptive system design are

- A system (linear in this case) with adaptive parameters
- The existence of a desired or target response d
- An optimality criterion (the MSE in this case) to be minimized
- A method (subsystem) to compute the optimal parameters

The method of least squares finds the optimal parameters (b, w) analytically. Our goal is to find alternate ways of computing the same parameters using a search procedure.

1.4.3 Analysis of the Error in the Space of the Parameters: The Performance Surface

Let us analyze the mean square error (J) as we change the parameters of the system (w and b). Without loss of generality, we are going to assume that $b = 0$ (or equivalently, that the mean of x and d has been removed), so that J becomes a function of the single variable w:

$$J = \frac{1}{2N}\sum_i (d_i - wx_i)^2 = \frac{1}{2N}\sum_i (x_i^2 w^2 - 2d_i x_i w + d_i^2) \tag{1.9}$$

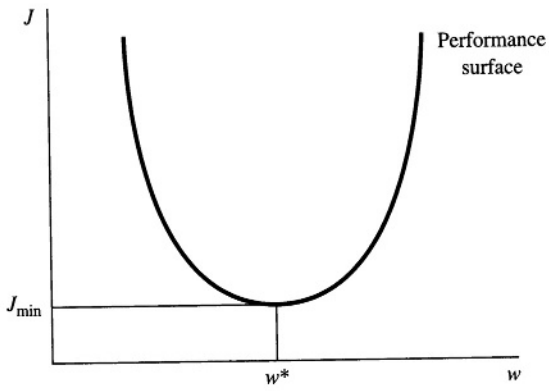


FIGURE 1-7 The performance surface for the regression problem

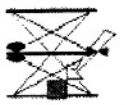
If w is treated as the variable and all other parameters are held constant, we can immediately see that J is quadratic on w with the coefficient of w^2 (i.e., x_i^2) being always positive. In the space of the possible w values, J is a parabola facing upward (J is always positive since it is a sum of squares). The function $J(w)$ is called the *performance surface* for the regression problem (Figure 1-7). The performance surface is an important tool that helps us visualize how the adaptation of the weights affects the mean square error.

NEUROSOLUTIONS EXAMPLE 1.4

Plotting the performance surface

The performance surface is just a plot of the error criterion (J) versus the value of the weights, so what we will do during the simulation is to vary the Synapse weight (which corresponds to the slope parameter of the linear regressor) between two appropriate values. We can imagine that the error will be minimum at an intermediate value of the weight and that it will increase for both lower and higher values.

To modify the Synapse weight incrementally, we attach a Linear Scheduler to the Synapse and place the Matrix Viewer on it so we can see how the weight is changing. To visualize the MSE, we bring another Scatter Plot to the L2 Criterion. This will allow us to plot the cost versus weight (performance surface).



Linear Scheduler on the Synapse

Run the example and see how the slope parameter of the linear PE affects the mean square error of the linear regressor. As we are going to see, the input and desired signals tremendously affect the shape of the performance surface. But how can we change the shape of the performance curve without touching the data files? Let us substitute for the L2 Criterion with the Lp Criterion. Go to Palettes and open the Error Criteria menu. Click on the Lp Criterion and bring the pointer to the Breadboard. Notice that the pointer changed to a stamper. If you left-click on the L2 Criterion component, the L2 Criterion is replaced by the Lp component, which computes a cost given by the p norm:

$$J = \frac{1}{p}\sum_i \varepsilon_i^p$$

By default the norm is $p = 5$. Run the simulation again. What do you see? Does the location of the minimum change appreciably? What about the shape of the performance surface? Do you now better understand the function of the cost criterion?

Using the performance surface, we can develop a geometric method for finding the value of w, here denoted by w^*, which minimizes the performance criterion. Previously (Eq. 1.5), we computed w^* by setting to zero the derivative of J with respect to w.

The *gradient of the performance surface* is a vector (with the dimension of w) that always points toward the direction of maximum J change and with a magnitude equal to the slope of the tangent of the performance surface (Figure 1-8). If you visualize the performance surface as a hillside, each point on the hill will have a gradient arrow that points in the direction of steepest *ascent* at that point, with larger magnitudes for steeper slopes. A ball rolling down the hill will always attempt to roll in the direction opposite to the gradient arrow (the steepest descent). The slope at the bottom is zero, so the gradient is also zero (that is the reason the ball stops there).

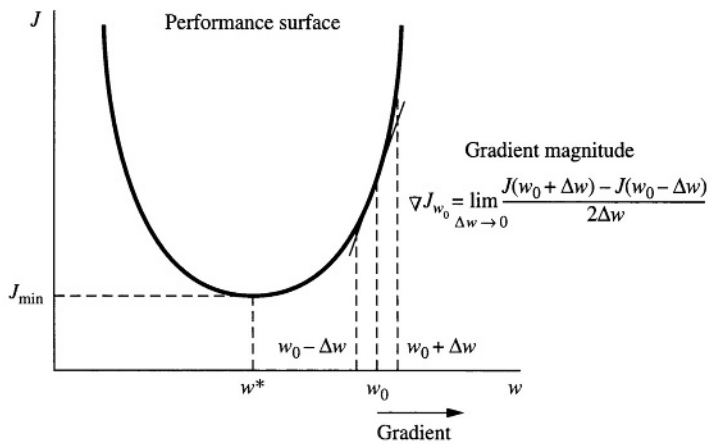


FIGURE 1-8 Performance surface and its gradient

Gradient Definition and Construction

The gradient is formally defined in terms of partial derivatives of a function $f(x, y)$. Let us consider a function $f(x, y)$ that has partial derivatives at x_0 and y_0. The gradient of f at x_0, y_0 is defined by

$$\text{grad } f(x_0, y_0) = \nabla f(x_0, y_0) = f_x(x_0, y_0)\mathbf{u}_x + f_y(x_0, y_0)\mathbf{u}_y$$

where $\mathbf{u}_x$ and $\mathbf{u}_y$ are the unit vectors along x and y, and f_x and f_y are the partial derivatives of f along the x and y directions, respectively, which are given by

$$f_x = \frac{\partial f(x, y)}{\partial x} \qquad f_y = \frac{\partial f(x, y)}{\partial y}$$

The gradient is associated with the concept of a directional derivative of a function. Let us assume we have a direction $\mathbf{u} = a\mathbf{u}_x + b\mathbf{u}_y$. The directional derivative of f at x_0, y_0 along $\mathbf{u}$ is

$$D_u f(x_0, y_0) = \lim_{h \to 0} \frac{f(x_0 + ha, y_0 + hb) - f(x_0, y_0)}{h}$$

The gradient can thus be defined as a function of the ordered derivatives:

$$D_u f(x_0, y_0) = (\text{grad } f(x_0, y_0)) \cdot \mathbf{u}$$

where the operation is the dot product of two vectors (for $\mathbf{v} = c\mathbf{u}_x + d\mathbf{u}_y$, $\mathbf{v} \cdot \mathbf{u} = ac + bd$).

This expression means that the maximum value of the directional derivative as a function of the direction $\mathbf{u}$ is given by the size of the gradient, and it occurs exactly when the direction $\mathbf{u}$ coincides with the gradient direction.

Moreover, we can also find this direction pretty easily. Let us consider the curve $C(x, y)$, defined as the line in the x, y plane where the function f has a constant value (this line is called the level curve or the contour of f). At a point x_0, y_0 in C the rate of change of f in the direction of the unit vector $\mathbf{u}$ tangent to C must be zero (see the preceding definition), that is,

$$D_u f(x_0, y_0) = (\text{grad } f(x_0, y_0)) \cdot \mathbf{u} = 0$$

But this implies that the gradient vector is perpendicular to the tangent vector $\mathbf{u}$ of the level curve at x_0, y_0. This explains the graphical construction outlined in the text.

In our special case the gradient has just one component along the weight axis w, $\nabla J = \nabla_w J$ given by

$$\nabla_w J = \frac{\partial J}{\partial w} \tag{1.10}$$

A graphical way to construct $\nabla_w J$ at a point w_0 is to first find the level curve (curve of constant J value) that passes through the point (also called the contour plot). Then take the tangent to the level curve at w_0. The gradient component $\nabla_w J$ is always perpendicular to the contour curve at w_0, with a magnitude given by the partial derivative of J with respect to the weight w (Eq. 1.10). For one weight (one-dimensional problem), as in Figure 1-8, the construction is simplified, and we have to find only the direction of the gradient on the axis.

At the bottom of the bowl, the gradient is zero, because the parabola has slope zero at the vertex. Thus, for a parabolic performance surface, computing the gradient and equating it to zero finds the value of the coefficients that minimize the cost, just as we did in Eq. 1.6. The important observation is that the analytical solution found by the least squares coincides with the minimum of the performance surface. Substituting the value of w^* into Eq. 1.9, the minimum value of the error ($J_{\min}$) can be computed.

More Properties of the Performance Surface

For a quadratic performance surface (Eq. 1.9), computing the gradient and equating it to zero finds the value of the coefficients that minimize the cost, that is,

$$\nabla J = \frac{\partial J}{\partial w} = 0 = \frac{1}{N}\left(-\sum_i d_i x_i + w \sum_i x_i^2\right) \tag{1B.7}$$

or

$$w^* = \frac{\sum_i x_i d_i}{\sum_i x_i^2} \tag{1B.8}$$

This solution is fundamentally the same as found in Eq. 1.6 ($b = 0$ is equivalent to assuming that the average values of x and d are zero). Substituting this value of w^* into Eq. 1.9, the minimum value of the error becomes

$$J_{\min} = \frac{1}{2N}\left[\sum_i d_i^2 - \frac{\left(\sum_i d_i x_i\right)^2}{\sum_i x_i^2}\right] \tag{1B.9}$$

Equation 1.9 can be rewritten in the form

$$J = J_{\min} + \frac{1}{2N}(w - w^*)\sum_i x_i^2 (w - w^*) \tag{1B.10}$$

To verify this, solve Eq. 1B.10 substituting Eq. 1B.8 for w^* and Eq. 1B.9 for $J_{\min}$. Notice the following observations:

- The minimum value of the error $J_{\min}$ (Eq. 1B.9) depends on both the input signal (x_i) and the desired signal (d_i).

- The location in coefficient space where the minimum w^* occurs (Eq. 1B.8) also depends on both x_i and d_i.
- The performance surface shape (Eq. 1B.10) depends only on the input signal (x_i).

NEUROSOLUTIONS EXAMPLE 1.5

Comparison of performance curves for different data sets

In this example we provide two sets of input files and two sets of output files. By changing the input data, we find that the minimum error, its location in the weight space (a simple line for this 1D example), and the shape of the performance surface change. On the other hand, if we change the desired signal, only the minimum value of the performance and its location change, but the overall shape remains the same.

1.4.4 Search of the Performance Surface with Steepest Descent

Since the performance surface is a paraboloid, which has a single minimum, an alternative procedure to find the best value of the coefficient w is to search the performance surface instead of computing the best coefficient analytically by Eq. 1.6. The search for the minimum of a function can be done efficiently using a broad class of methods based on gradient information. The gradient has two main advantages for the search:

- The gradient can be computed locally.
- The gradient always points in the direction of maximum change.

If the goal is to reach the minimum, the search must be in the direction opposite to the gradient. Thus the overall method of searching can be stated in the following way.

Start the search with an arbitrary initial weight $w(0)$, where the iteration number is denoted by the index in parentheses. Then compute the gradient of the performance surface at $w(0)$, and modify the initial weight proportionally to the negative of the gradient at $w(0)$. This changes the operating point to $w(1)$. Then compute the gradient at the new position $w(1)$, and apply the same procedure again; that is,

$$w(k+1) = w(k) - \eta \nabla J(k) \tag{1.11}$$

where η is a small constant and $\nabla J(k)$ denotes the gradient of the performance surface at the kth iteration. The constant η is used to maintain stability in the search by ensuring that the operating point does not move too far along the performance surface. This search procedure is called the *steepest descent* method. Figure 1-9 illustrates the search procedure.

If we trace the path of the weights from iteration to iteration, intuitively we see that if the constant η is small, eventually the best value for the coefficient w^* will be found. Whenever $w > w^*$, we decrease w, and whenever $w < w^*$, we increase w.

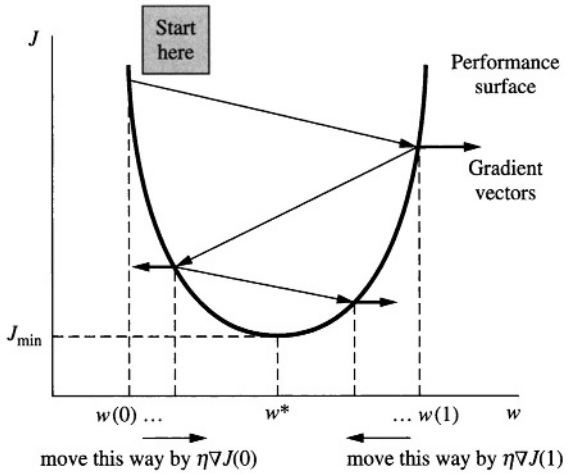


FIGURE 1-9 The search using the gradient information

1.5 ESTIMATION OF THE GRADIENT: THE LMS ALGORITHM

An adaptive system can use the gradient to optimize its parameters. The gradient, however, is usually not known explicitly and thus must be estimated. Traditionally, the difference operator is used to estimate the derivative, as outlined in Figure 1-8. A good estimate, however, requires many small perturbations to the operating point to obtain a robust estimation through averaging. The method is straightforward but not very practical.

In the late 1960s Widrow[2] proposed an extremely elegant algorithm to estimate the gradient that revolutionized the application of gradient descent procedures. His idea is very simple: Use the instantaneous value as the estimator for the true quantity. For our problem, this means to drop the summation in Eq. 1.9 and define the gradient estimate at step k as its instantaneous value. Substituting Eq. 1.4 into Eq. 1.10, removing the summation, and then taking the derivative with respect to w yields

$$\nabla J(k) = \frac{\partial}{\partial w(k)} J = \frac{\partial}{\partial w(k)} \frac{1}{2N} \sum_i \varepsilon_i^2 \approx \frac{1}{2} \frac{\partial}{\partial w(k)} (\varepsilon^2(k)) = -\varepsilon(k)x(k) \qquad \textbf{(1.12)}$$

What Eq. 1.12 tells us is that an instantaneous estimate of the gradient at iteration k is simply the product of the current input to the weight times the current error. The amazing thing is that the gradient can be estimated with one multiplication per weight. This is the gradient estimate that led to the famous *least mean square (LMS) algorithm* (or LMS rule). The estimate will be noisy, however, since the algorithm uses the error

[2]Bernard Widrow was one of the first researchers to explore engineering applications of adaptive systems. We are going to hear a lot about him in this book.

from a single sample instead of summing the error for each point in the data set (e.g., the MSE is estimated by the error for the current sample). But remember that the adaptation process does not find the minimum in one step. Normally, many iterations are required to find the minimum of the performance surface, and during this process the noise in the gradient is being averaged (or *filtered*) out.

If the estimator of Eq. 1.12 is substituted in Eq. 1.11, the steepest descent equation becomes

$$w(k+1) = w(k) + \eta\varepsilon(k)x(k) \tag{1.13}$$

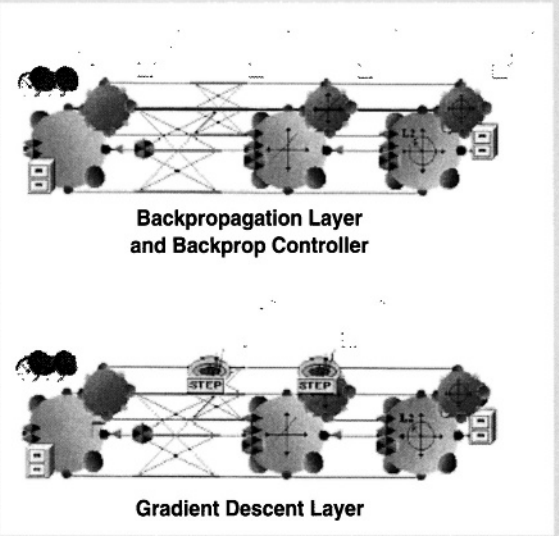
This equation is the *LMS algorithm.* With the LMS rule we do not need to worry about perturbation and averaging to properly estimate the gradient at each iteration; it is the iterative process that is improving the gradient estimator. The small constant η is called the *step size,* or the *learning rate.*

NEUROSOLUTIONS EXAMPLE 1.6

Adapting the linear PE with LMS

Several things have to be added to the previous Breadboard of the linear PE to make it learn automatically using the LMS algorithm. The methodology will be explained in more detail later. However, the technique used in NeuroSolutions is called back-propagation. In short, the algorithm passes the input data forward through the network and the error information (desired − output) backward through another network. The error is propagated through a second network, which can be obtained from the original network with minor and well-established modifications (more about this later). Thus at every component there is a local activity (x) and a local error (ε) such that the weights of the network can be modified by Eq. 1.13. NeuroSolutions implements this technique by adding two additional layers to the network: the backpropagation layer and the gradient-search layer. These two layers can be automatically added to the Breadboard. The backpropagation layer looks like a small version of the network that sits on top of the original network (in red instead of orange). The gradient-search layer sits on top of the backpropagation layer and uses one of the gradient-search methods to adjust the weights. In this case, the gradient-search layer is a simple "step" layer, which implements the gradient descent rule of Eq. 1.13. Notice that only the components that have adjustable weights [the Synapse (w) and Bias Axon (b)] have gradient-search components.

In addition to the two layers, we need an additional Controller to manage the back-propagation layer. The Backprop Controller sits above the yellow Controller and sets parameters such as whether we use batch or on-line learning. In this example we use batch learning; that is, the learning algorithm will compute all the weight updates for the training set, add them up, and at the end of the epoch (one presentation of all the training data) update the weights according to Eq. 1.13. The value of the step size will be set at 0.01 and the training will use 200 iterations.



When you run the network, watch the regression line move toward the optimal value in the Scatter Plot. When the simulation stops, notice that the weight is approximately 0.139, the bias is approximately 1.33, and the error is approximately 0.033—all in excellent agreement with the optimal values we computed analytically.

You should explore this Breadboard by entering several values of the step size and opening the Inspector to see how each component is configured.

1.5.1 Batch and Sample-by-Sample Learning

The LMS algorithm was presented in a form in which the weight updates are computed for each input sample and the weights modified after each sample. This procedure is called sample-by-sample learning, or *on-line training.* As we have mentioned, the estimate of the gradient is going to be noisy; that is, the direction toward the minimum is going to zigzag around the gradient direction.

An alternative solution is to compute the weight update for each input sample and store these values (without changing the weights) during one pass through the training set, which is called an *epoch.* At the end of the epoch, all the weight updates are added together, and only then will the weights be updated with the composite value. This method adapts the weights with a cumulative weight update, so it will follow the gradient more closely. This method is called the *batch training mode,* or *batch learning.* Batch learning is also an implementation of the steepest-descent procedure. In fact, it provides an estimator for the gradient that is smoother than the LMS. We will see that the agreement between the analytical quantities that describe adaptation and the ones obtained experimentally is excellent with the batch update.

Batch versus On-line Learning

The on-line and batch modes are slightly different, although both will perform well for parabolic performance surfaces. One major difference is that the batch algorithm keeps the system weights constant while computing the error associated with each sample in the input. Since the on-line version is constantly updating its weights, its error calculation (and thus gradient estimation) uses different weights for each input sample. This means that the two algorithms visit different sets of points during adaptation. However, they both converge to the same minimum.

Note that the number of weight updates of the two methods for the same number of data presentations is very different. The on-line method (LMS) does an update each sample, while batch does an update each epoch, that is,

$$\text{LMS updates} = (\text{batch updates}) \times (\#\ \text{of samples in training set})$$

The batch algorithm is also slightly more efficient in terms of number of computations.

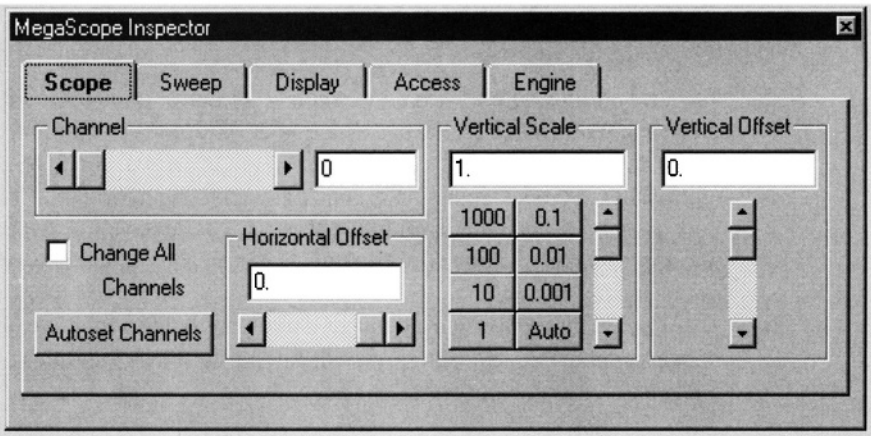
To visualize the differences between these two update methods, we will plot the value of the cost during adaptation (called the *learning curve*).

NEUROSOLUTIONS EXAMPLE 1.7

Batch versus on-line adaptation

It is important to visualize the differences in adaptation for on-line and batch learning. Up to now we have been using the batch mode. In this example we set the Backprop Controller to use on-line training. To display the learning curve, we have to introduce one new component: the MegaScope. The MegaScope is a probe, that acts just like an oscilloscope: It plots a continuous stream of inputs, using the iteration number as the x axis. The MegaScope sits on top of the Data Storage component.

The important controls of the MegaScope are the scales of the x and y axes. In the scope level of the Inspector, we can select the vertical scale (y axis) and the offset of each channel. Alternatively, we can use the autoscale feature. The horizontal scale is selected in the sweep level of the Inspector (number of samples per division). Remember that the number of samples displayed is defined by the Data Storage component.



To create the learning curve, we simply place a Data Storage component over the L2 Criterion and then place a MegaScope on top of it. We will see that the learning curve is not smooth anymore because we are updating the weights after each example. Since the individual errors vary from sample to sample, our updates will make the learning curve noisy. The learning curve will have a periodic component superimposed on a decaying exponential. The exponential tells us that we are approaching a better overall solution. The periodic features show the error obtained for each input sample, while the envelope is related to the learning curve for the batch mode. Note that the weights never stabilize; the performance curve should otherwise be smooth and converge to a single final value. Since there is more noise in on-line learning, we must decrease the step size to get smoother adaptation. We recommend using a step size 10 times smaller than the step size for batch learning. But the price paid is a longer adaptation time; the system needs more iterations to get to a predefined final error. Experiment with the learning rates to observe this behavior.

1.5.2 Robustness and System Testing

One of the interesting aspects of the LMS solution is its robustness. From the explanation given (Figure 1-9), no matter what the initial condition for the weights, the solution always converges to basically the same value. We can even add some noise to the desired response and find out that the linear regressor parameters are basically unchanged. This robustness is rather important for real-world problems, where noise is omnipresent.

The group of input samples and desired responses (shown in Table 1-1) used to train the system are called collectively the training set for obvious reasons. It is with their information that the system parameters were adapted. But once the optimal parameters are found, the parameters should be fixed. When the system is utilized for new inputs never encountered before, it will produce for each input a response based on the parameters obtained during training. If the new data comes from the same experiment, the response should resemble the value of the desired response for that particular input value.

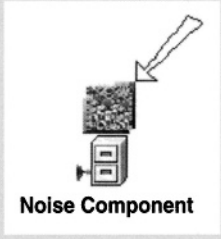
Thus we see that the system has the ability to extrapolate responses for new data. This is an important feature, since in general we wish the performance obtained in the training set to also apply (generalize) to the new data when the system is deployed. But due to the methodology utilized to derive the parameter values, we can never be exactly sure of how well the system will respond to new data.

For this reason it is a good methodology to use a test set to verify the system performance before deploying it in the real-world application. The test set consists of new data not used for training but for which we still know the desired response. It is like the final rehearsal before a play's inauguration. We should also compute the correlation coefficient in the test set. Normally, we will find a slight decrease in performance from the training set. If the performance in the test set is not acceptable, we have to go back to the drawing board. When this happens in regression, the most common problem is that the training data is inadequate—either in quantity or in exhaustive coverage of the experimental conditions. This point will be addressed in more depth in the following chapters.

NEUROSOLUTIONS EXAMPLE 1.8

Robustness of LMS to noise

The LMS algorithm is very robust. It works from any arbitrary location and even works well with noise added to the desired data. To demonstrate that the system works well even with noisy data, we add one additional component to the Breadboard from the previous example: the Noise component. The Noise component allows uniform, Gaussian, or "user-defined" noise to be added to the input or desired signals. We will add the Noise component to the desired signal and watch as the system moves close to the optimal location even with the noisy data.



1.5.3 Computing the Correlation Coefficient in Adaptive Systems

The correlation coefficient, r, tells how much of the variance of d is captured by a linear regression on the independent variable x. As such, r is a very powerful quantifier of the modeling result. It has a great advantage with respect to the MSE because it is automatically normalized, while the MSE is not. However, the correlation coefficient

is blind to differences in means because it is a ratio of variances (see Eq. 1.8); that is, as long as the desired data and input covary, r will be small, in spite of the fact that they may be far apart in actual value. Thus we need both quantities (r and MSE) when testing the results of regression.

Although the correlation coefficient can be computed directly from x and d (Eq. 1.8), we would like to estimate r at the output of the linear system to follow the adaptive systems' methodology. From Eq. 1B.4, we can write

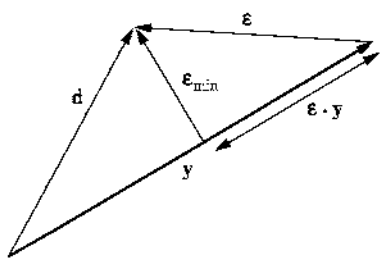
$$\tilde{r} = \frac{\sqrt{\sum_i (y_i - d)^2}}{\sqrt{\sum_i (d_i - d)^2}} \tag{1.14}$$

Note, however, that y changes during adaptation, so we should wait until the system adapts to read the final correlation coefficient ($\tilde{r} \rightarrow r$ when $w \rightarrow w^*$). During adaptation the numerator of Eq. 1.14 can be larger than the denominator, giving a value for r larger than 1, which is meaningless. We therefore propose to compute a new parameter $\tilde{r}$ that is a reasonable proxy for the correlation coefficient, even during adaptation. We subtract a term from the numerator of Eq. 1.14 that becomes zero at the optimal setting but limits $\tilde{r}$ so that its value is always between -1 and 1, even during adaptation. We can write

$$\tilde{r} = \frac{\sqrt{\sum_i (y_i - \bar{d})^2} - \dfrac{\sum_i \varepsilon_i (y_i - \bar{d})}{\sqrt{\sum_i (y_i - \bar{d})^2}}}{\sqrt{\sum_i (d_i - d)^2}} \tag{1.15}$$

Computation of the Correlation Coefficient

It is important to remember (Eq. 1B.6) that with the optimal coefficients the regression error $\boldsymbol{\varepsilon}_{\min}$, interpreted as a vector, is perpendicular to the Adaline output $\mathbf{y}$. This condition is called the orthogonality condition. In fact, from the following figure it is easy to see that the smallest error is obtained when the projection of $\mathbf{d}$ on $\mathbf{y}$ is the orthogonal projection.



During adaptation the error will always be larger than $\varepsilon_{\min}$, meaning that $\mathbf{y}$ can be larger than $\mathbf{d}$, so Eq. 1.14 may be larger than 1, which is misleading since $r < 1$. Using the fact that the minimum error is perpendicular to $\mathbf{y}$, we can compute the dot product of $\boldsymbol{\varepsilon}$ with

y and subtract it from the numerator of Eq. 1.14. We can prove that this new numerator is always smaller than **d** and that the dot product is zero at the optimal solution and so will not affect the final value of the correlation coefficient. This is exactly what is done in Eq. 1.14.

Note that all these quantities can be computed on-line with the information of the error, the output, and the desired response. Remember, however, that Eq. 1.15 measures the correlation coefficient only when the Adaline has been totally adapted to the data.

NEUROSOLUTIONS EXAMPLE 1.9

Estimating the correlation coefficient during learning

NeuroSolutions does not include a component to compute the correlation coefficient. It does, however, allow you to write your own components. These custom components are called DLLs. A custom component looks just like the component it takes the place of, except that its icon has "DLL" printed on it. In this example we include a custom component to compute the correlation coefficient. This component looks exactly like an L2 Criterion component, except it has "DLL" printed on it.

Plug in the values of the optimal weights and verify that the formula Eq. 1.15 gives the correct correlation coefficient. Slightly modify w to 0.120 and verify that the correlation coefficient decreases. If you plug in values for w and b that are very far away from the fitted regression, this estimation of r using Eq. 1.14 becomes less accurate, but is still bound by -1 and 1. The example also uses LMS to adapt the coefficients. Observe that the correlation coefficient is always between -1 and 1 during adaptation and that the final value corresponds to the computed one.

1.6 A METHODOLOGY FOR STABLE ADAPTATION

During adaptation the learning algorithm automatically changes the system parameters using Eq. 1.13. This adaptation algorithm has one free parameter (the step size) that must be user selected. To appropriately set the step size, the user should have a good understanding of what is happening inside the system. In this section we quantify the adaptation process and develop visualization tools that will help you understand how well the system is learning.

1.6.1 Learning Curve

As is readily apparent from Figure 1-9, when the weights approach the optimal value, the values of $J(k)$ (the MSE at iteration k) will also decrease, approaching its minimum value $J_{\min}$. One of the best ways to monitor the convergence of the adaptation process is to plot the error at each iteration. The plot of the MSE across iterations is called the *learning curve* (Figure 1-10). The learning curve is as important for adaptive systems

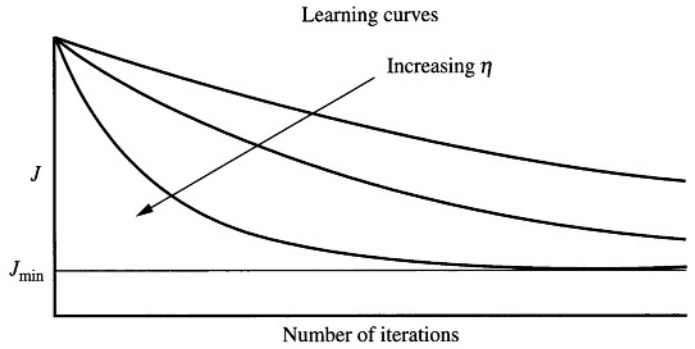


FIGURE 1-10 The learning curve

as the thermometer is to check your health. It is an external, scalar, easy-to-compute indication of how well the system is learning. But similar to body temperature, it is unspecific; that is, when the system is not learning, it does not tell us why.

Notice that the error approaches the minimum in a one-sided manner (i.e., always larger than $J_{\min}$). As you can expect, the rate of decrease of the error depends on the value of the step size η. Larger step sizes will take fewer iterations to reach the neighborhood of the minimum, provided that the adaptation converges. However, too large a step size creates a divergent iterative process, and the optimal solution is not obtained. We therefore must seek a way to find the largest possible step size that guarantees convergence.

NEUROSOLUTIONS EXAMPLE 1.10

The learning curve

The goal of this example is to display the learning curve and to show how the learning rate affects its shape. This example plots the mean square error during adaptation, which we called the learning curve, the thermometer of learning.

When you run the simulation, watch how the error moves toward zero as the regression line moves toward the optimal location. You can also change the learning rates and watch the regression line moving more quickly or slowly toward the optimal location, thus causing the learning curve to be steeper or shallower. The visualization of the regression line contains more information about what the system is doing but is very difficult to compute and display in higher dimensions. The learning curve, however, is an external, scalar quantity that can be easily measured with minimal overhead.

1.6.2 Weight Tracks

An adaptive system modifies its weights in an effort to find the best solution. The plot of the value of a weight over time is called the *weight track.* Weight tracks are an important

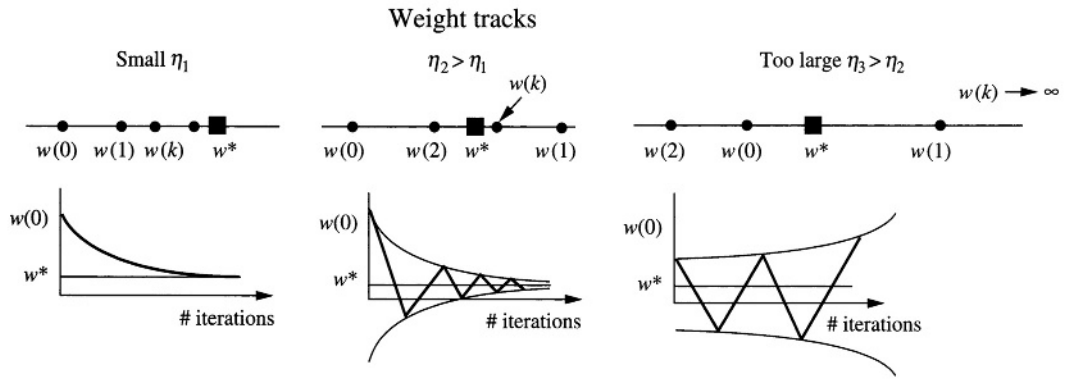


FIGURE 1-11 Weight tracks and plots of the weight values across an iteration for three values of η

and direct measure of the adaptation process. The problem is that normally our system has many weights, and we don't know what their optimal values are. Nevertheless, the dynamics of learning can be inferred and monitored from the weight tracks.

In the gradient-descent adaptation, adjustments to the weights are governed by two quantities (Eq. 1.11): the step size η and the value of the gradient at the point. Even for a constant step size, the weight adjustments become smaller and smaller as the adaptation approaches w^*, since the slope of the quadratic performance surface is decreasing near the bottom of the performance surface. Thus the weights approach their final values asymptotically (Figure 1-11).

Three cases are depicted in Figure 1-11. If the step size is small, the weight converges monotonically to w^*, and the number of iterations to reach the bottom of the bowl may be large. If the step size η is increased, the convergence will be faster but still monotonic. After η reaches a value called *critically damped,* the weight will approach w^* in an oscillatory fashion ($\eta_2 > \eta_1$); that is, it will overshoot and undershoot the final solution. The number of iterations necessary to reach the neighborhood of w^* will start increasing again. If the step size is too large ($\eta_3 > \eta_2$), the iterative process will diverge; that is, instead of getting closer to the minimum, the search will visit points of larger and larger MSE until there is a numeric overflow. We say that the learning diverged.

NEUROSOLUTIONS EXAMPLE 1.11

Weight tracks

It is very instructive to observe the linear PE parameters during learning and how they change as a function of the step size. Let us install a MegaScope over the Synapse to visualize the slope parameter of the regressor and over the Bias Axon to visualize the regressor bias. These are called weight tracks. Run the simulation and watch how changing the step sizes affects the way the system approaches its final weights.

The weight tracks are a finer display of how adaptation is progressing, but the problem is that in systems with many weights it becomes impractical to observe all the weight tracks. Why do we say that weight tracks give us a better handle on the adaptation parameters? Enter 0.02 for the step size and see the weight tracks converge monotonically to their minimum value. Now enter 0.035. The weight tracks are oscillating toward the final value, which means that the system is already in the underdamped regime (but the learning curve is still monotonically decreasing toward the minimum at a faster rate). We can expect divergence if we increase the weights further. Try 0.038 and see it happen. Relate this behavior to Figure 1-11.

1.6.3 Largest Step Size for Convergence

As we have just discussed, we would like to choose the largest step size possible for fastest convergence without creating an unstable system. Since adjustment to the weights is a product of the step size and the local gradient of the performance surface, it is clear that the largest step size depends on the shape of the performance surface. We saw already (Eq. 1B.10) that the shape of the performance surface is controlled by the input data, so we can conclude that the maximum step size will be dictated by the input data. But how?

If we rewrite and manipulate the equations that produce the weight values in terms of the first weight $w(0)$, we get

$$w(k+1) = w^* + (1-\eta\lambda)^{k+1}(w(0) - w^*) \tag{1.16}$$

where

$$\lambda = \frac{1}{N}\sum_i x_i^2$$

The term $(1-\eta\lambda)^k$ must be less than or equal to 1 to guarantee weight convergence (and less than 1 to guarantee convergence to zero, giving the solution $w(k+1) = w^*$). This implies that

$$|\rho| = |1-\eta\lambda| < 1 \Rightarrow \eta < \eta_{\max} = \frac{2}{\lambda} \tag{1.17}$$

where ρ is the geometric ratio of the iterative process. Hence the value of the step size η must always be smaller than $2/\lambda$. The fastest convergence is obtained with the critically damped step size of $1/\lambda$. The closer η is to $1/\lambda$, the faster is the convergence, but faster convergence also means that the iterative process is closer to instability. We can visualize this in Figure 1-11. When η is increased, a monotonic (overdamped) convergence to w^* is substituted by an alternating (underdamped) convergence that finally degenerates into divergence.

Derivation of the Largest Step Size

The best way to find the upper bound for η is to write the equation that produces the weight values. Let us rewrite the ideal performance surface (Eq. 1B.10) as

$$J = J_{\min} + \frac{\lambda}{2}(w - w^*)^2 \tag{1B.11}$$

where

$$\lambda = \frac{1}{N}\sum_i x_i^2 \tag{1B.12}$$

By computing the gradient of J (Eq. 1B.11), we get

$$\nabla J = \lambda(w - w^*) \tag{1B.13}$$

so the iteration that produces the weight updates (Eq. 1.11) can be written as

$$w(k+1) = (1 - \eta\lambda)w(k) + \eta\lambda w^* \tag{1B.14}$$

This is a first-order linear constant-coefficient difference equation, which can be solved by induction. First, let us subtract w^* from both sides to yield

$$w(k+1) - w^* = (1 - \eta\lambda)(w(k) - w^*)$$

Start with a solution $w(0)$,

$$\begin{aligned} w(1) - w^* &= (1 - \eta\lambda)(w(0) - w^*) \\ w(2) - w^* &= (1 - \eta\lambda)^2(w(0) - w^*) \\ w(3) - w^* &= (1 - \eta\lambda)^3(w(0) - w^*) \end{aligned}$$

which provides by induction the equation

$$w(k) = w^* + (1 - \eta\lambda)^k(w(0) - w^*)$$

There is a slight practical problem that must be solved. During batch learning the weight updates are added together during an epoch to obtain the new weight. This effectively includes a factor of N in the LMS weight update formula, Eq. 1.13. To apply the analysis of the largest step size of Eq. 1.17, we have to use a normalized step size:

$$\eta_n = \frac{\eta}{N} \tag{1.18}$$

With this modification, even if the number of samples in our experiment changes, the step sizes do not need to be modified. Note that for on-line learning ($N = 1$) we get the LMS rule again. We will always use normalized step sizes, but to make the notation simpler, we will drop the subscript n in the normalized step size. An added advantage of using normalized step sizes is that we can switch between on-line updates and batch updates without having to change the step size in the simulations.

The analysis of the largest step size of Eq. 1.17 also applies *in the mean* to the LMS algorithm. However, since LMS uses an instantaneous (noisy) estimate of the gradient, even when η obeys Eq. 1.17, instability may occur. When the iterative process diverges, the algorithm "forgets" its location in the performance surface; that is, the values of the weights change drastically. This means that all the iterations up to that point were wasted. Hence with LMS it is common to include a safety factor of 10 in the largest η ($\eta < 0.1\eta_{\max}$) or to use batch training which reduces the noise in the estimate of the gradient.

1.6.4 Time Constant of Adaptation

An alternative view of the adaptive process is to quantify the convergence of $w(k)$ to w^* in terms of an exponential decrease. We know that $w(k)$ converges to w^* as a geometric progression (Eq. 1.16). The envelope of the geometric progression of weight values can be approximated by an exponential decay $\exp(-t/\tau)$, where τ is the *time constant of weight adaptation*. A single iteration or epoch can be considered a unit of time. One may want to know approximately how many iterations are needed until the weights converge. The time constant of weight adaptation can be written as

$$\tau = \frac{1}{\eta\lambda} \tag{1.19}$$

which clearly shows that fast adaptation (small time constant τ) requires large step sizes. For all practical purposes the iterative process converges after four time constants.

Derivation of the Time Constant of Weight Adaptation

Writing $\exp(-1/\tau) = \rho$ and expanding the exponential in Taylor series,

$$\rho = \exp\left(-\frac{1}{\tau}\right) = 1 - \frac{1}{\tau} + \frac{1}{2!\tau^2} - \cdots$$

we get approximately $\rho \approx 1 - 1/\tau$. We saw that geometric ratio of the gradient descent is Eq. 1.17, so we get

$$\tau = \frac{1}{\eta\lambda}$$

The steps used to derive the time constant of weight adaptation can be applied also to come up with a closed-form solution to the decrease of the cost across iterations, which is called the *time constant of adaptation.* Equation 1.16 tells us how the weights converge to w^*. If the equation for the weight recursion is substituted in the equation for the cost (Eq. 1B.11) we get

$$J = J_{\min} + \lambda(1 - \eta\lambda)^{2k}(w(0) - w^*)^2$$

which means that J also approximates $J_{\min}$ in a geometric progression, with a ratio equal to ρ^2. Therefore the time constant of adaptation is

$$\tau_{\text{mse}} = \frac{\tau}{2}$$

Since the geometric ratio is always positive, J approximates $J_{\min}$ monotonically (i.e., an exponential decrease). The time constant of adaptation describes the learning time (in number of iterations) needed to adapt the system in a practical way. Notice that these expressions assume that the adaptation follows the gradient. With the instantaneous estimate used in the LMS, J may oscillate during adaptation, since the estimate is noisy. But even in the LMS, J will approach $J_{\min}$ in a one-sided way (i.e., always greater than or equal to $J_{\min}$).

NEUROSOLUTIONS EXAMPLE 1.12

Linear regression without bias

The previous example solved the linear regression problem with one weight and one bias. To compare the equations previously given (which are a function of a single parameter) with the simulations, we have to make a modification in the data set or in the simulation. Shortly we will see how to extend the analysis for multiple weights, but for the time being let us work with the simpler case.

We replace the Bias Axon with an Axon, a component that simply adds its inputs, so the regression solution becomes $y = wx$, which has to pass through the origin. With this new Breadboard we can compare the numerical results of the simulations directly with all the equations derived in this section, since there is only one free parameter. Batch updates are used throughout.

The optimal value of the slope parameter is computed by Eq. 1.5, which gives $w = 0.30009$, with an average error of 0.23. This solution is different from the value obtained previously ($w = 0.139511$) for the bias regressor because the regression line is now constrained to pass through the origin. It turns out that this constrained solution is worse than before, as we can see by the error (0.23 versus 0.033). Observing the output (red points) and the input samples (blue) in the Scatter Plot shows clearly what we are describing.

Computing λ by Eq. 1.16 yields 54, so according to Eq. 1.17 the maximum step size is $\eta = 3.6 \times 10^{-2}$. The critically damped solution is obtained with a step size

of 1.8×10^{-2}, and adaptation with a step size below this value is overdamped. When we run the simulator in the overdamped case, the weights approach the final value monotonically; for the critically damped case they stabilize quite rapidly, while for the underdamped case they oscillate around the final value, and the convergence takes more iterations. Notice also that the linear regressor "vibrates" around the final position, since the slope parameter is overshooting and undershooting the optimal value.

According to Eq. 1.19 the critically damped step size $\tau = 1$, so the solution should stabilize in four updates (epochs). This step size yields the fastest convergence. Go to the Controller Inspector and use the epoch button to verify the number of samples until convergence.

1.6.5 Rattling

Up to now our main focus was the speed of adaptation, that is, how fast the weights approximate w^* or, equivalently, how fast J approximates $J_{\min}$. Unfortunately, this is only part of the story. For fast convergence we need large step sizes (η). But when the search is close to the minimum w^*, where the gradient is small but not zero, the iterative process continues to wander around a neighborhood of the minimum solution without ever stabilizing. This phenomenon is called *rattling* (Figure 1-12), and the rattling basin increases proportionally to the step size η. This means that when the adaptive process is stopped by an external command (such as the number of iterations through the data), the weights may not be exactly at w^*. We know they are in the neighborhood of the minimum point but not exactly at the optimum.

If we picture the performance surface (Figure 1-12) when the final weights are not at w^*, there will be a penalty in performance; that is, the final MSE will be higher than $J_{\min}$. In the theory of adaptation the difference between the final MSE and the $J_{\min}$ (normalized by $J_{\min}$) is called the *misadjustment M*.

$$M = \frac{J_{\text{final}} - J_{\min}}{J_{\min}} \tag{1.20}$$

This means that in search procedures that use gradient descent there is an intrinsic compromise between accuracy of the final solution (a small misadjustment) and speed

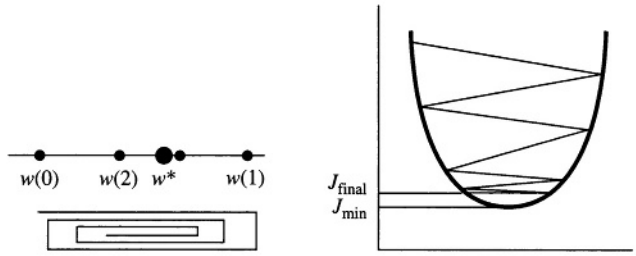


FIGURE 1-12 Rattling of the iteration procedure

of convergence. The parameter that controls this compromise is the step size η. High η means fast convergence but also large misadjustment, while small η means slow convergence but little misadjustment.

NEUROSOLUTIONS EXAMPLE 1.13

Rattling

We observed in Example 1.7 how noisy the learning curve became with the on-line update. This is an external indication that the weights were changing from sample to sample even after the system reached the neighborhood of the optimum. The implication of this random movement in the weights is a penalty in the final MSE. In this example we show and exactly quantify the rattling.

The rattling has important consequences for adaptation, since if we set the step size large for fast convergence, we pay a price of inaccurate coefficients, which are translated into an excess MSE. The rule of thumb for LMS is to use a step size that is 1/10 of the largest possible step size. For a step size close to the largest possible, the MSE for the epoch is effectively smaller than the theoretical minimum, which is impossible. This happens because the parameters are changing so much with each update that the slope is continuously varying. The problem is that when we stop the training, we do not know whether the final value of the weight is a good approximation to the theoretical regression line.

This shows that for adaptive systems the final MSE is only part of the story. We have to make sure that the system coefficients have stabilized. It is interesting to note that with batch updates there is no rattling, so in the linear case the batch solution is more appropriate. Observe this in the simulations by displaying the MSE for large and small step sizes. We are just paying the small price of storing the individual weight updates. For nonlinear systems the batch is unfortunately no longer always superior to the on-line update, as we will see.

This example shows that obtaining a small MSE is a necessary but not sufficient condition for stable adaptation. *Adaptation also requires that the weights of the model settle onto stable values*. This second condition is required because the system can endlessly change its parameters to fit the present sample. This will always give a small MSE, but from a modeling point of view it is a useless solution because *no single model is found to fit the data set.*

1.6.6 Scheduling the Step Sizes

As we saw in the latest examples, for fast convergence to the neighborhood of the minimum, a large step size is desired. However, the solution with a large step size suffers from rattling. One attractive solution is to use a large learning rate in the beginning of training to move quickly toward the location of the optimal weights, but then the learning rate should be decreased to obtain good accuracy on the final weight values. This is

called *learning rate scheduling*. This simple idea can be implemented with a variable step size controlled by

$$\eta(k+1) = \eta(k) - \beta \tag{1.21}$$

where $\eta(0) = \eta_0$ is the initial step size, and β is a small constant. Note that the step size is being linearly decreased at each iteration. If we have control of the number of iterations, we can start with a large step size and decrease it to practically zero toward the end of training. The value of β needs to be experimentally determined. Alternatively, we can decrease the step size slowly (in optimization this slow decrease is called annealing) using a linear, geometric, or logarithmic rule.

More on Scheduling Step Sizes

If the initial value of η_0 is set too high, the learning can diverge. The selection of β can be even trickier than the selection of η because it is highly dependent on the performance surface. If β is too large, the weights may not move quickly enough to the minimum and the adaptation may stall. If β is too small, the search may reach the global minimum quickly and must wait a long time before the learning rate decreases enough to minimize the rattling. There are other (more automatic) methods for adapting the learning rate that we discuss later in the book.

NEUROSOLUTIONS EXAMPLE 1.14

Scheduling of step sizes

In this demonstration we vary the step size using the scheduling component already introduced in Example 1.4. The scheduler is a component that takes an initial value from the component it is attached to, and changes the value according to a predetermined rule. Here we use the linear rule, and since we want to decrease the step size, the factor β is negative. We should set a maximum and a minimum value just to make sure that the parameters are always within the range we want. β should be set according to the number of iterations and the initial and final values ($\mu_{\text{Init}} - \beta N = \mu_{\text{Resid}}$).

Here the important parameter is the minimum (the residual step size is set at 0.001), because after scheduling we want to let the system fine-tune its parameters to find the best approximation of the minimum. However, notice that this implies that the parameter is already in its optimal neighborhood, and this depends on a lot of unknown factors. Thus if the scheduling is not right, the adaptation may stall in positions far from the minimum.

You should explore the Breadboard by entering other values for β and the final value and see their impact on the final weight value. You can also use the exponential and the logarithmic schedulers and see how they behave. Which one do you prefer for this case?

1.7 REGRESSION FOR MULTIPLE VARIABLES

Assume that d is now a function of several inputs $x_1, x_2, \ldots, x_D$ (independent variables), and the goal is to find the best linear regressor of d on all the inputs (Figure 1-13). For $D = 2$ this corresponds to fitting a plane through the N input samples or a hyperplane in the general case of D dimensions.

As an example, let us assume that we have two variables, x_1 (speed) and x_2 (feed rate), that affect the surface roughness (d) of a machined workpiece. In abstract units the values of x_1, x_2, and d for 15 workpieces are presented in Table 1-2. The goal is to

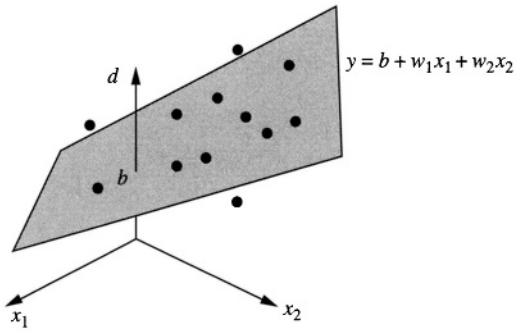


FIGURE 1-13 Fitting a regression plane to a set of samples in 2D space

TABLE 1-2 Multiple Regression Data

x_1	x_2	d
1	2	2
2	5	1
2	3	2
2	2	2
3	4	1
3	5	3
4	6	2
5	5	3
5	6	4
5	7	3
6	8	4
7	6	2
8	4	4
8	9	3
9	8	4

find how well one can explain the quality of machining by the two variables x_1 and x_2 and which is the most important parameter.

As before, we assume that the measurements $\mathbf{x}$ are noise free and that $\mathbf{d}$ is contaminated by a noise vector $\boldsymbol{\varepsilon}$ with these properties: Gaussian distributed with components that are zero mean, of equal variance σ^2, and uncorrelated with the inputs. The regression equation when $D = 2$ is now

$$\varepsilon_i = d_i - (b + w_1 x_{i1} + w_2 x_{i2}) \tag{1.22}$$

where x_{i1} is the ith value of x_1 (the ith workpiece in the training set). In the general case we write Eq. 1.22 as

$$\varepsilon_i = d_i - \left(b + \sum_{k=1}^{D} w_k x_{ik}\right) = d_i - \sum_{k=0}^{D} w_k x_{ik} \qquad i = 1 \ldots N \tag{1.23}$$

where we made $w_0 = b$ and $x_{i0} = 1$ (compare with Eq. 1.3). The goal of the regression problem is to find the coefficients $w_0, \ldots, w_D$. To simplify the notation we will put all these values into a vector $\mathbf{w} = [w_0, \ldots, w_D]^T$ that minimizes the MSE of ε_i over the N samples. Figure 1-14 shows that the linear PE now has D inputs and one bias.

The MSE becomes for this case

$$J = \frac{1}{2N} \sum_i \left(d_i - \sum_{k=0}^{D} w_k x_{ik}\right)^2 \tag{1.24}$$

The solution to the extreme (minimum) of this equation can be found in exactly the same way as before, that is, by taking the derivatives of J with respect to the unknowns (w_k) and equating the result to zero.

This yields a set of $D+1$ equations in $D+1$ unknowns called the *normal equation*

$$\sum_i x_{ij} d_i = \sum_{k=0}^{D} w_k \sum_i x_{ik} x_{ij} \qquad j = 0, 1, \ldots, D \tag{1.25}$$

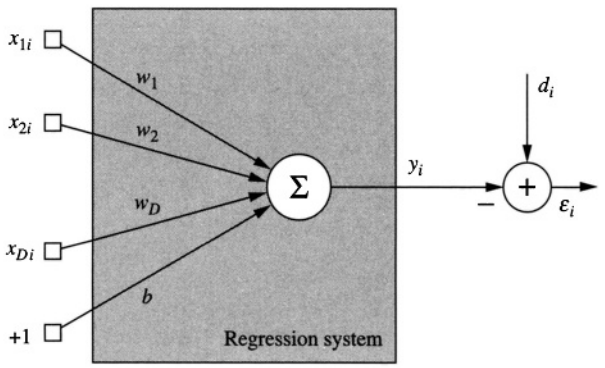


FIGURE 1-14 Regression system for multiple inputs

Derivation of Normal Equations

When the derivative of J with respect to the unknown quantities (the weights) is taken, we end up with a set of $D + 1$ equations in $D + 1$ unknowns:

$$\frac{\partial J}{\partial w_j} = -\frac{1}{N}\sum_i x_{ij}\left(d_i - \sum_{k=0}^{D} w_k x_{ik}\right) = 0 \qquad \text{for } j = 0, \ldots, D \tag{1B.15}$$

The solution is the famous normal matrix equation:

$$\sum_i x_{ij} d_i = \sum_{k=0}^{D} w_k \sum_i x_{ik} x_{ij} \qquad j = 0, 1, \ldots, D \tag{1B.16}$$

or expanding

$$\begin{cases} \sum_i x_{i0} d_i = \sum_k w_k \sum_i x_{ik} x_{i0} \\ \sum_i x_{i1} d_i = \sum_k w_k \sum_i x_{ik} x_{i1} \\ \vdots \\ \sum_i x_{iD} d_i = \sum_k w_k \sum_i x_{ik} x_{iD} \end{cases} \tag{1B.17}$$

Notice that these equations are linear in the unknowns (the w_j), so they can be easily solved.

The normal equations can be written much more compactly with matrix notation (see Appendix A, Section 3). Let us define

$$R_{kj} = \frac{1}{N}\sum_i x_{ik} x_{ij} \tag{1.26}$$

as the *autocorrelation* of the input samples for indices k and j. As you can see, the autocorrelation measures similarity across the samples of the training set. When $k = j$, $\mathbf{R}$ is just the sum of the squares of the input samples (the variance if the data is zero mean). When k differs from j, $\mathbf{R}$ measures the sum of the cross products for every possible combination of the indices. As we did for $\mathbf{w}$, we will also put all these R_{kj} values into a matrix $\mathbf{R}$, that is,

$$\mathbf{R} = \begin{bmatrix} R_{00} & \cdots & R_{0D} \\ \cdots & \cdots & \cdots \\ R_{D0} & \cdots & R_{DD} \end{bmatrix}$$

Thus we obtain pairwise information about the structure of the data set.

Let us call

$$P_j = \frac{1}{N}\sum_i x_{ij} d_i \tag{1.27}$$

the *cross-correlation* of the input x for index j and desired response d, which can also be put into a vector $\mathbf{p}$ of dimension $D-1$. P_j measures the similarity between the input x and the desired response d at shift j. Substituting these definitions in Eq. 1.25, the set of normal equations can be written simply

$$\mathbf{p} = \mathbf{R}\mathbf{w}^* \qquad \text{or} \qquad \mathbf{w}^* = \mathbf{R}^{-1}\mathbf{p} \tag{1.28}$$

where $\mathbf{w}$ is a vector with the $D+1$ weights w_i. $\mathbf{w}^*$ represents the value of the $D+1$ weights for the optimal (minimum) solution. $\mathbf{R}^{-1}$ denotes the inverse of the autocorrelation matrix (see Appendix A, Section 3.11). Equation 1.28 states that the solution of the multiple regression problem can be computed analytically as the product of the inverse of the autocorrelation matrix of the input samples and the cross-correlation vector between the input and the desired response. The least square solution for this problem yields

$$y = 1.353480 + 0.286191x_1 - 0.004195x_2$$

It is remarkable that we are able to write an equation that describes the relationship between the two variables when only measured data samples were given. This attests to the power of linear regression. But as for the single variable case, we still do not know how accurately the equation fits the data, that is, how much of the variance of the input is actually captured by the regression model. The multiple correlation coefficient r_m can also be defined in the multiple-dimensional case for a single output as

$$r_m = \sqrt{\frac{\mathbf{w}^{*T}\mathbf{U}_x\mathbf{d} - Nd^2}{\mathbf{d}^T\mathbf{d} - Nd^2}} \tag{1.29}$$

and measures the amount of variation explained by the linear regression, normalized by the variance of $\mathbf{d}$. In this expression $\mathbf{d}$ is the vector built from the desired responses d_i, and $\mathbf{U}$ is a matrix whose columns are the input data vectors. For this case $r_m = 0.68$, so there is a large portion of the variability that is not explained by the linear regression. (Either the process is nonlinear, or there are more variables involved.) We can still approximate the correlation coefficient for the multiple regression case by Eq. 1.15 after the system has adapted.

Multiple-Variable Correlation Coefficient

The idea of the correlation coefficient is the same for one or multiple dimensions. The equations get a little more complicated, since we are now working with an ensemble of input vectors, so the nice form of Eq. 1.8 has to be modified. An ensemble of vectors is better

described as a matrix, so we are going to define a new matrix **U** as

$$\mathbf{U}_x = \begin{bmatrix} x_1^1 & \cdots & x_1^N \\ \cdots & \cdots & \cdots \\ x_D^1 & \cdots & x_D^N \end{bmatrix}$$

where each column is one of the input samples. We are likewise going to define a column vector **d** with all the desired responses (this is a vector for the single-output regression, otherwise it also becomes a matrix):

$$\mathbf{d} = \begin{bmatrix} d_1 \\ \cdots \\ d_N \end{bmatrix}$$

The total error variance can be written as

$$\boldsymbol{\varepsilon}^T\boldsymbol{\varepsilon} = \mathbf{d}^T\mathbf{d} - \mathbf{w}^{*T}\mathbf{U}_x\mathbf{d}$$

where $\mathbf{w}^*$ is the set of optimal coefficients. This expression can be easily derived if the output of the regressor is substituted in the definition of the error [Dunteman, 1984]. The part of the error that is explained by the linear model is the second term. The variance of the output is expressed in the same way (just subtract the mean of the desired signal). Thus if we normalize this equation by the variance of the desired response, we get

$$r^2 = \frac{\mathbf{w}^{*T}\mathbf{U}_x\mathbf{d} - N\bar{d}^2}{\mathbf{d}^T\mathbf{d} - N\bar{d}^2}$$

which leads to the correlation coefficient for the multivariate case.

NEUROSOLUTIONS EXAMPLE 1.15

Multivariable regression

Moving to multi-dimensional inputs is very simple in NeuroSolutions. You simply change the input and desired files (for the new input data) and change the input Axon to accept two inputs. The rest is automatic. In this example we do all this for you using macros. Note that in the two-dimensional case the regression line is now a regression plane. In NeuroSolutions, there is not currently a good way of showing a plane in three dimensions, so we will not plot our regression plane. When we run the network, we will see that the learning curve (one of our only indications of whether the network is training correctly) decreases steadily and that the weights eventually approach the theoretical optimal weights.

The amazing thing about the adaptive system's methodology is that we changed the problem, but the technique to solve it did not change significantly. It is true that we have to dimension the system properly and choose new values for the step size, but the fundamental aspects of the methodology did not change at all.

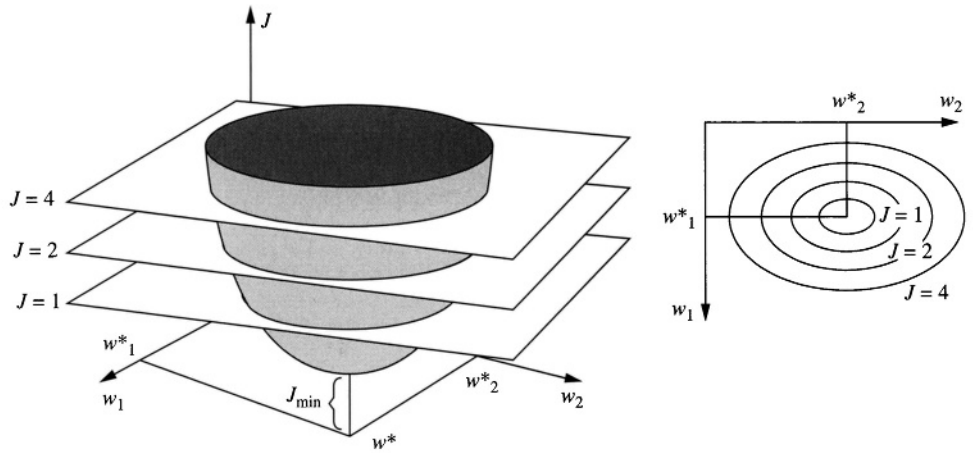


FIGURE 1-15 The performance surface for two dimensions and its contour plot

1.7.1 Setting the Problem as a Search Procedure

All the concepts previously mentioned for linear regression can be extended to the multiple regression case. The performance surface concept can be extended to multiple dimensions, making J a paraboloid in $D + 1$ dimensions, facing upward. (Figure 1-15 depicts the two weight-case.) J now involves matrix computations, but it remains a scalar quantity that is a quadratic function of the weights.

$$J = \left[0.5\mathbf{w}^T\mathbf{R}\mathbf{w} - \mathbf{p}^T\mathbf{w} + \sum_i \frac{d_i^2}{2N}\right] \tag{1.30}$$

where the superscript T means the transpose.

The coefficients that minimize the solution are

$$\nabla J = 0 = \mathbf{R}\mathbf{w}^* - \mathbf{p} \quad \text{or} \quad \mathbf{w}^* = \mathbf{R}^{-1}\mathbf{p} \tag{1.31}$$

which gives exactly the same solution as Eq. 1.28. In the space (w_1, w_2), J is a paraboloid facing upward.

Derivation of the Optimal Solution

Let us just take the derivative of J (Eq. 1.30) with respect to the weights, using matrix operations:

$$\frac{\partial J}{\partial \mathbf{w}} = 0.5\mathbf{R}\mathbf{w} + 0.5\mathbf{w}^T\mathbf{R} - \mathbf{p} = \mathbf{R}\mathbf{w} + \mathbf{R}^T\mathbf{w} - 2\mathbf{p} = 2\mathbf{R}\mathbf{w} - 2\mathbf{p}$$

since the transpose of **R** is equal to itself because of its Toeplitz structure. If we equate this to zero, we obtain the optimal weights:

$$\mathbf{w}^* = \mathbf{R}^{-1}\mathbf{p}$$

which is the equation in the text.

NEUROSOLUTIONS EXAMPLE 1.16

Checking the LMS solution with the optimal weights

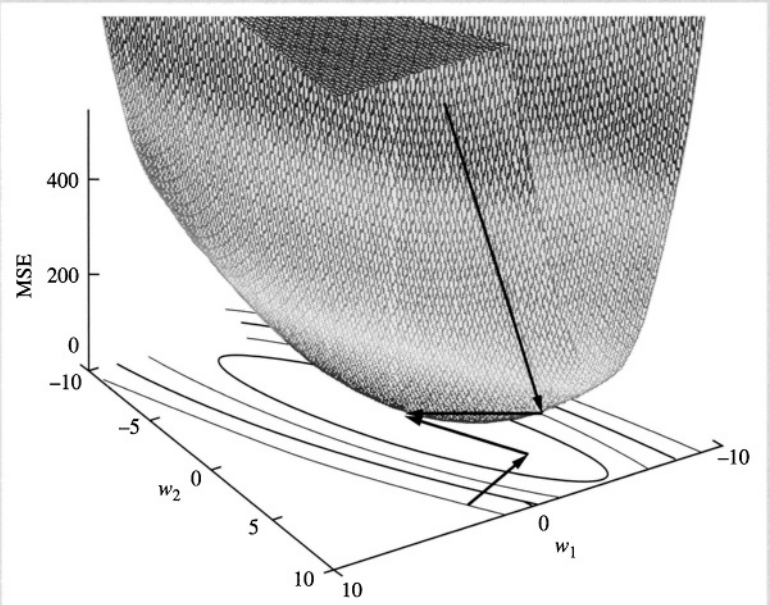
Let us first consider a least square solution with only two weights, w_1 and w_2, since we can still compute it easily by hand. For the data set of Table 1-2, the autocorrelation matrix is (from Eq. 1.26)

$$\mathbf{R} = \frac{1}{15}\begin{bmatrix} 416 & 429 \\ 429 & 490 \end{bmatrix}$$

To determine the eigenvalues, we solve the equation

$$\det[\mathbf{R} - \lambda\mathbf{I}] = 0$$

which yields $\lambda_1 = 59$ and $\lambda_2 = 1.5$. From these results we can immediately see that the eigenvalue spread is roughly 40, so the performance surface paraboloid is very skewed (i.e., much narrower in one direction). The performance surface is shown in the following figure. Notice how it is very steep in one direction and very shallow



in the other. Thus, if we train the network with gradient descent, we would expect it to move very quickly down the steep slope at first and then to move slowly down the valley toward the optimum.

To compute the optimal solution, we first need to compute the cross-correlation vector (from Eq. 1.27):

$$\mathbf{p} = \frac{1}{15}\begin{bmatrix}212\\229\end{bmatrix}$$

For the two-dimensional case it is still easy to solve for w_1 and w_2 by writing (from Eq. 1.28)

$$\begin{cases}416w_1 + 429w_2 = 212\\429w_1 + 490w_2 = 229\end{cases}$$

which gives for optimal weights $w_1 = 0.2848$ and $w_2 = 0.2180$. The minimum J is 0.390. When we run the simulator with the Axon replacing the Bias Axon (no bias), the network weights approach these values.

Performance Surface Properties

The minimum value of the error can be obtained by substituting the optimal weight (Eq. 1.31) into the cost equation (Eq. 1.30), yielding

$$J_{\min} = \frac{1}{2}\left[\sum_i \frac{d_i^2}{N} - \mathbf{p}^T\mathbf{w}^*\right] \qquad \textbf{(1B.18)}$$

We can rewrite the performance surface in terms of its minimum value and $\mathbf{w}^*$ as

$$J = J_{\min} + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^T\mathbf{R}(\mathbf{w} - \mathbf{w}^*) \qquad \textbf{(1B.19)}$$

For the one-dimensional case this equation is the same as Eq. 1B.10 ($\mathbf{R}$ becomes a scalar equal to the variance of the input). In the space (w_1, w_2) J is now a paraboloid facing upward. The shape of J is again solely dependent on the input data (through its autocorrelation function). One can show that

- The principal axes of the performance surface contours (surfaces of equal error) correspond to the *eigenvectors of the input correlation matrix* $\mathbf{R}$ *(see Appendix A).*
- The eigenvalues of $\mathbf{R}$ give the rate of change of the gradient along the principal axis of the surface contours of J (Figure 1-16).

The eigenvectors and eigenvalues of the input autocorrelation matrix are all that matters to understand convergence of the gradient descent in multiple dimensions. The eigenvectors represent the natural (orthogonal) coordinate system to study the properties of $\mathbf{R}$.

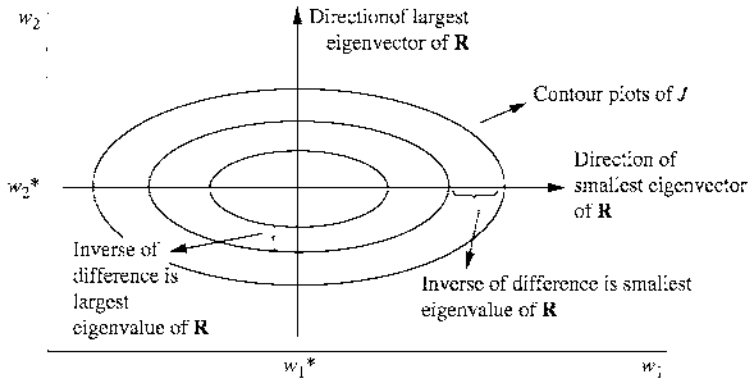


FIGURE 1-16 Contour plots of the performance surface with two weights

In fact, along these coordinates the convergence of the algorithm can be studied as a joint adaptation of several (one for each dimension of the space) unidimensional algorithms. Along each eigenvector direction (the axes of the ellipsoids) the algorithm behaves just like the one-variable case that we studied in the beginning of this chapter. The eigenvalue becomes the projection of the data onto that direction, just as λ in Eq. 1B.11 is the projection of the data on the weight direction. But in any other direction the adaptation is coupled.

The location of the performance surface in weight space depends on both the input and desired response (Eq. 1.31). The minimum error also depends on both (Eq. 1B.18). Multiple regression finds the location of the minimum of a paraboloid placed in an unknown position in weight space. The input distribution defines *the shape of the performance surface*. The input distribution and its relation with the desired response distribution define both the *value* of the minimum of the error and the *location in coefficient space* where that minimum occurs.

As in the one-dimensional case, the autocorrelation of the input (**R**) completely specifies the shape of the performance surface (Eq. 1B.19). However, the location of the performance surface in the space of the weights (Eq. 1.31) and its minimum value (Eq. 1B.18) depend also on the desired response.

1.7.2 Steepest Descent for Multiple Weights

Gradient techniques can also be used to find the minimum of the performance surface, but now the gradient is a vector with $D + 1$ components

$$\nabla \mathbf{J} = \left[\frac{\partial J}{\partial w_0}, \ldots, \frac{\partial J}{\partial w_D} \right]^T \tag{1.32}$$

The extension of Eq. 1.11 is

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \eta \nabla \mathbf{J}(k) \tag{1.33}$$

where all quantities are vectors, that is, $\mathbf{w}(k) = [w_0(k), \ldots, w_D(k)]^T$. To calculate the largest step size η, we again rewrite the update equation in the form of (see the box "Derivation of Largest Step Size" and compare with Eq. 1B.14)

$$\mathbf{w}(k+1) = (\mathbf{I} - \eta\mathbf{R})\mathbf{w}(k) + \eta\mathbf{R}\mathbf{w}^* \tag{1.34}$$

where $\mathbf{I}$ is the identity matrix, $\mathbf{R}$ is the input autocorrelation matrix, and $\mathbf{w}^*(k) = [w_0^*(k), \ldots, w_D^*(k)]^T$. The solution of this equation is *cross coupled;* that is, the way $\mathbf{w}$ converges to $\mathbf{w}^*$ depends on the behavior of the geometric progression in all the $D + 1$ directions. Therefore the simple picture of having $\mathbf{w}(k+1)$ converge to $\mathbf{w}^*$ with a single geometric ratio, as in the unidimensional case, has to be modified. One can show that the weights converge with different time constants, each related to an eigenvalue of $\mathbf{R}$.

Convergence for Multiple-Weights Case

One can show that the condition to guarantee convergence [Widrow and Stearns 1985] is

$$\lim_{k \to \infty} (\mathbf{I} - \eta\mathbf{\Lambda})^k = 0 \tag{1B.20}$$

where $\mathbf{\Lambda}$ is the eigenvalue matrix,

$$\mathbf{\Lambda} = \begin{bmatrix} \lambda_0 & \cdots & 0 \\ \cdots & \cdots & \cdots \\ 0 & \cdots & \lambda_D \end{bmatrix} \tag{1B.21}$$

which means that in every principal direction of the performance surface (given by the eigenvectors of the input correlation matrix $\mathbf{R}$), we must have

$$0 < \eta < \frac{2}{\lambda_i} \tag{1B.22}$$

where λ_i is the corresponding eigenvalue. This equation also means that with a single η each weight $w_i(k)$ is approaching its optimal value w_i^* with a different time constant ("speed"), so the weight tracks bend, and the path is no longer a straight line toward the minimum.

This is the mathematical description of our earlier statement that the gradient descent algorithm behaves like many one-dimensional univariable algorithms along the eigenvector directions. Notice that Eq. 1B.21 is diagonal, so there is no cross coupling between time constants along the eigenvector directions.

In any other direction of the space there will be coupling. However, we can still decompose the overall weight tract as a combination of weight tracts along each eigendirection, as we did in Figure 1.16. Eq. 1B.22 shows that the step size along each direction obeys the same rule as the unidimensional case (Eq. 1.17).

1.7.3 Step Size Control

As we have seen, the set of values taken by the weight during adaptation is called the weight track. The weight moves in the opposite direction of the gradient at each point, so the weight track depicts the gradient direction at each point of the performance surface visited during adaptation. Therefore the gradient direction tells us about the performance surface shape. In particular, it is useful to construct the contour plot of J since the gradient has to be perpendicular to the lines that link points with the same J value. The contour plot provides a graphical construction for the gradient at each point of the performance surface.

Given a point in a contour, we take the tangent of the contour at that point. The gradient is perpendicular to the tangent, so the weights will move along the gradient line and point in the opposite direction. Likewise, if we run the adaptation algorithm with several initial conditions and we record the value of J at each point, we can determine the contour plots by taking ellipses that pass through the points of equal cost and are perpendicular to the weight tracks.

When the *eigenvalues* of $\mathbf{R}$ are the same (see Appendix A, Section 3.17), the contour plots are circular, and the gradient always points to the center, that is, to the minimum. In this case the gradient descent has only a single time constant as in the one-dimensional case. But this is an exceptional condition. In general, the eigenvalues of $\mathbf{R}$ will be different. When the eigenvalues are different, the weight track bends because it follows the direction of the gradient at each point, which is perpendicular to the contours (Figure 1-17). The gradient direction no longer points to the minimum, which means that the weight tracks will not be straight lines to the minimum. The adaptation will take longer for two reasons. First, a longer path to the minimum will be taken. Second, the step size must be decreased compared with the circular case. Let us address the step-size aspect further.

For guaranteed convergence the learning rate in each ith principal direction of the performance surface must be

$$0 < \eta < \frac{2}{\lambda_i} \tag{1.35}$$

where λ_i is the corresponding eigenvalue. The worst case condition to guarantee convergence to the optimal $\mathbf{w}^*$ in all directions is therefore

$$\eta < \frac{2}{\lambda_{max}} \tag{1.36}$$

That is, the step size η must be smaller than the inverse of the largest eigenvalue of the autocorrelation matrix. Otherwise the iteration will diverge in one or more directions. Since the adaptation is coupled, divergence in one direction will cause the entire system to diverge.

In the early stages of adaptation the convergence is primarily along the direction of the largest eigenvalue, since the weight update along this direction will be bigger.

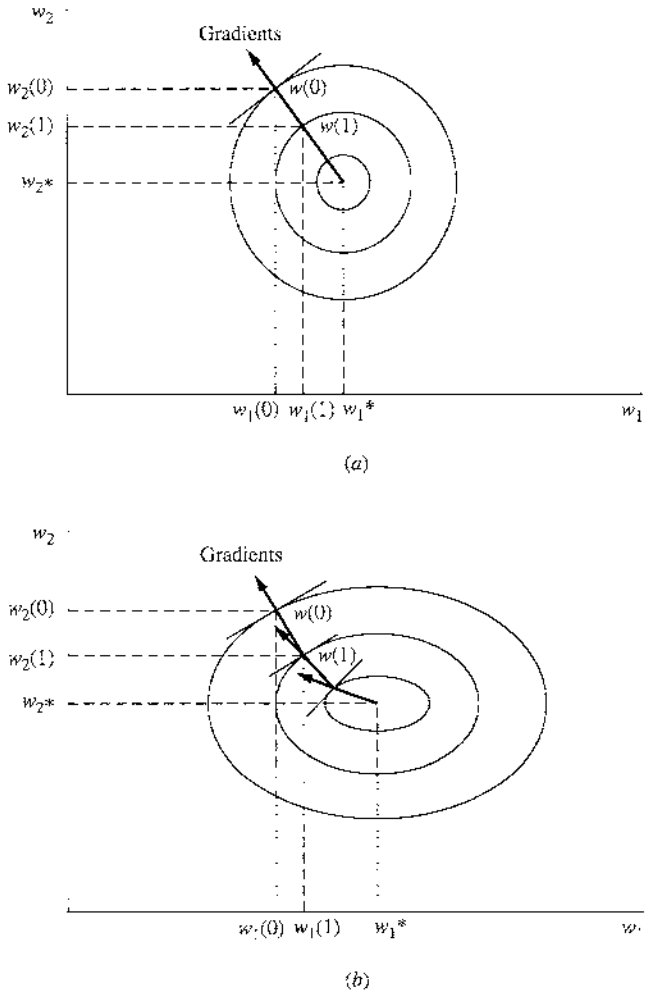


FIGURE 1-17 Weight track toward the minimum: (a) equal eigenvalues; (b) unequal eigenvalues

On the other hand, toward the end of adaptation the algorithm will mainly adapt only the weight associated with the smallest eigenvalue (which corresponds to the smallest time constant). The time constant of adaptation is therefore

$$\tau = \frac{1}{\eta \lambda_{\min}} \tag{1.37}$$

An implication of this analysis is that when the *eigenvalue spread* of $\mathbf{R}$ is large, there will be very different time constants of adaptation in each direction. This reasoning gives a clear picture of the fundamental constraint of adapting the weights using

gradient descent with a single step size η: The speed of adaptation is controlled by the smallest eigenvalue, while the largest step size is constrained by the inverse of the largest eigenvalue. This means that if the ratio between the largest and the smallest eigenvalues (the eigenvalue spread) is large, the convergence will be intrinsically slow. This problem cannot be avoided when only a single step size is used in the steepest descent.

The learning curve will approach $J_{\min}$ in a geometric progression as before. However, *there will be many different time constants of adaptation, one for each direction.* Initially, the learning curve will decrease at the rate of the largest eigenvalue, but toward the end of adaptation the rate of decrease of J is controlled by the time constant of the smallest eigenvalue.

Estimation of Eigenvalue Spread

The eigenvalue spread can be computed by an eigendecomposition of **R**, but this is a time-consuming operation and is hardly ever performed. An estimate of the eigenvalue spread for the multidimensional-data case is the ratio between the maximum and the minimum of the magnitude of the Fourier transform of the input data.

Alternatively, simple inspection of the correlation matrix of the input can provide an estimation of the time to find a solution. The best possible case is when **R** is diagonal, with equal values in the diagonal, because in this case the eigenvalue spread is 1 and the gradient descent travels in a straight line to the minimum. We cannot have a faster convergence than this, even when second-order methods (such as Newton's method, studied later) are used. When **R** is diagonal but with different values, the ratio of the largest number over the smallest is a good approximation to the eigenvalue spread. When **R** is fully populated, the analysis becomes much more difficult.

NEUROSOLUTIONS EXAMPLE 1.17

Visualizing the weight tracks and speed of adaptation

According to our previous calculations, the largest step size for convergence is (from Eq. 1.36) 3.3×10^{-2}. The critically damped mode along the largest eigenvector should be 1.6×10^{-2}. The time constant of adaptation for the largest step size is around 20 iterations (epochs for batch); that is, the convergence should take 80 epochs with this step size.

When we run the simulations, the algorithm converges first along the direction of the largest eigenvalue (largest eigenvector direction) and then along the direction of the smallest eigenvector. Since the eigenvalue spread is 40, the steps are much bigger along the largest eigenvector direction. If we look at the figure of Example 1.16, we can see that the weights converge perpendicular to the contour plots since this is the steepest descent path. As we will see, there are two distinct regions in the learning curve: In the beginning it is controlled by the geometric ratio along the largest eigenvector, while toward the end it is controlled by the geometric ratio of the smallest eigenvector.

After running this example and observing the weight tracks, let us change the input data file to a data set with a smaller eigenvalue spread. Click the input file icon and bring up its Inspector by clicking the right mouse button. Remove the present input file, and add the file regression2a.asc from the *Data\ Chapters\ CH1* folder on the CD-ROM.

The modification was made only in the variable x_2; all the rest is the same. Respond to the panel *associate* by clicking on the *close* button. In the *customize* panel, skip the desired signal, and click on *close*. You have just modified the input data to this example. This new file has a much smaller eigenvalue spread, so we can expect that the weight tracks are basically straight lines to the minimum. Compute the new eigenvalue spread, and adjust the learning rates so that the convergence is as fast as possible.

1.7.4 The LMS Algorithm for Multiple Weights

It is straightforward to extend the gradient estimation given by the LMS algorithm from one dimension to many dimensions. We just apply the instantaneous gradient estimate Eq. 1.12 to each element of Eq. 1.33. The LMS for multiple dimensions reads

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \eta\varepsilon(k)\mathbf{x}(k) \tag{1.38}$$

An interesting feature is that the LMS *adaptation rule still uses local computations;* that is, for the ith weight we can write

$$w_i(k+1) = w_i(k) + \eta\varepsilon(k)x_i(k) \tag{1.39}$$

Note that although the analysis of the gradient descent techniques became complex, the LMS algorithm itself is still very simple. This is one reason why LMS is so widely used. But since LMS is a steepest-descent algorithm, the analysis and discussions concerning the largest step size for convergence and coupling of modes also apply to the LMS algorithm.

NEUROSOLUTIONS EXAMPLE 1.18

Visualizing weight tracks with on-line learning

In this example we configure the Backprop Controller to on-line learning to implement the LMS algorithm. Notice that the weight tracks follow basically the same path as before, but now the path is much more irregular due to the sample-by-sample update of the weights. When the eigenvalue spread is very large (the performance surface is very steep in one direction and shallow in others), the problem is difficult for LMS to solve. Any small perturbation in the smallest eigenvector direction gets amplified by the large eigenvalue spread.

1.7.5 Multiple Regression with Bias

Up to now we have implemented and solved analytically the multiple regression problem without bias. The reason for this is only for simplicity. With two weights we can still easily solve the multiple regression case by hand; however, if the bias is added, we must do the computations with three parameters. The simulations are transparent to these difficulties, since we just substitute the Axon by a Bias Axon. Note that the largest step size between the two cases differs since the input data is effectively changed if we interpret the bias as a weight connected to an extra constant input of one. Hence the autocorrelation function and its eigenvalue spread change.

We should state that the use of a bias is called the full least square solution, and it is the recommended way to apply least squares. The reason can be understood easily: When a bias is utilized in the PE, the regression line is not restricted to pass through the origin of the space, and smaller errors normally are achieved. There are two equivalent ways to solve the full least squares problem for D input variables:

- The input and desired responses need to be modified so that they become zero-mean variables. (This is called the deviation or z scores.) In this case a D weight regression will effectively solve the original problem. The bias b is computed indirectly by

$$b = \overline{d} - \sum_{i=1}^{N} w_i \overline{x}_i$$

  where w_i are the optimal weights and the bars represent mean values.
- Alternatively, the input matrix has to be extended with an extra column of 1s (the first column). This transforms $\mathbf{R}$ into a $(D+1)\times(D+1)$ matrix, which introduces a $D+1$ weight in the solution (the bias).

NEUROSOLUTIONS EXAMPLE 1.19

Linear regression without bias

We will now substitute a Bias Axon for the Axon in the previous Breadboard. This will effectively provide the regression solution without constraining the regression plane to pass through the origin. We see that the weight tracks are very similar in the beginning but that the error continues to drop, and the weights advance toward the $w_1 = 0$ line. This means that the optimal solution changed. We now have a better solution than before but with increased complexity of the performance surface (four dimensions instead of three) and an increased number of adjustable parameters in our system (two weights and a bias).

1.7.6 The LMS Algorithm in Practice

We can use some rules of thumb to choose the step size in the LMS algorithm. The step size should be normalized by the variance of the input data estimated by the trace of **R**.

$$\eta = \frac{\eta_0}{tr(\mathbf{R})} \tag{1.40}$$

where $\eta_0 = 0.1$ to 0.01. This normalization by the input variance was the original rule proposed by Widrow to adapt the Adaline. We can expect the algorithm to converge (settling time) in a number of iterations k given by four times the time constant of adaptation:

$$k \approx 4\tau_{mse} = \frac{2}{\eta\lambda_{min}} \tag{1.41}$$

The LMS algorithm has a misadjustment that is basically one half the trace of **R** times η:

$$M = \frac{\eta}{2} tr(\mathbf{R}) \tag{1.42}$$

When the eigenvalues are equal, the misadjustment can be approximated by

$$M \approx \frac{D + 1}{4\tau_{mse}}$$

which allows us to give the following rule of thumb: The misadjustment equals the number of weights divided by the settling time, or equivalently, selecting η so that it produces 10 percent misadjustment means a training duration in iterations of 10 times the number of inputs.

1.8 NEWTON'S METHOD

If you are familiar with numerical analysis, you may be asking why we aren't using Newton's method for the search. Newton's method is known to find the roots of quadratic equations in one iteration. The minimum of the performance surface can be equated to the root of the gradient equation Eq. 1.32, as outlined by Eq. 1.31. Hence Newton's method can also be used in the search. The adaptive weight equation using Newton's method is

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \mathbf{R}^{-1}\nabla J(k) \tag{1.43}$$

Compare this with Eq. 1.33 and note that the gradient information is weighted by the inverse of the correlation matrix of the input and that η is equal to 1. This means that Newton's method corrects the direction of the search so that it always points to

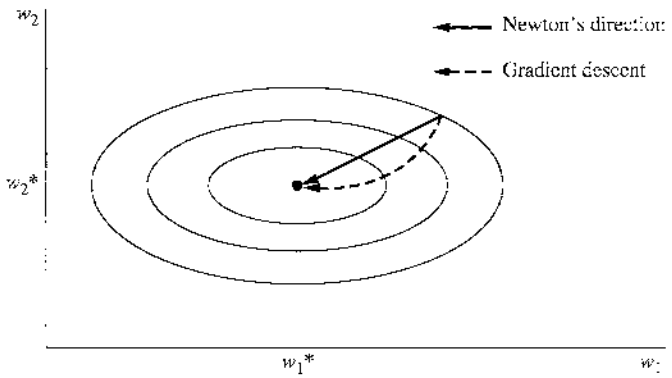


FIGURE 1-18 Directions of the steepest descent and Newton's method

the minimum, while the gradient descent points to the maximum direction of change. These two directions may or may not coincide (Figure 1-18).

They coincide when the contour plots are circles, that is, when the largest and the smallest eigenvalue of the correlation matrix are the same. When the ratio of the largest to the smallest eigenvalue (*the eigenvalue spread*) increases, the slope of the performance surface in the two directions differs more and more. Thus for large eigenvalue spreads the optimization path taken by gradient descent is normally much longer than the path taken by Newton's method. This implies that Newton's method will be faster than LMS when the input data correlation matrix has a large eigenvalue spread.

Newton's Derivation

The equation can be easily proved if we recall the gradient of the performance surface:

$$\nabla J = \mathbf{R}\mathbf{w} - \mathbf{p}$$

Left-multiply by $\mathbf{R}^{-1}$ to obtain

$$\mathbf{R}^{-1}\mathbf{p} = \mathbf{w} - \mathbf{R}^{-1}\nabla J$$

and then substitute in the optimal solution (Eq. 1.28) to obtain

$$\mathbf{w}^* = \mathbf{w} - \mathbf{R}^{-1}\nabla J$$

From this equation we can derive the incremental equation presented previously.

Another advantage of Newton's method versus the steepest descent method is in the time constant of adaptation. When the gradient is multiplied by $\mathbf{R}^{-1}$, not only is the direction of the gradient being changed, but also *the different eigenvalues in each*

direction are being equalized. What this means is that Newton's method is automatically correcting the time constant of adaptation for each direction so that *all the weights converge at the same rate.* Hence Newton's method has a single time constant of adaptation, unlike the steepest descent method.

These advantages of Newton's method should not come as a surprise, because Newton's method uses much more information about the performance surface (the curvature). In fact, to implement Newton's method, you need to compute the inverse of the correlation matrix, which takes significantly longer than the single multiplication required by the LMS method and also requires global information. Newton's method is also brittle; that is, if the surface is not exactly quadratic, the method may diverge. This is the reason Newton's method is normally modified to also include a small step size η instead of using $\eta = 1$ as in Eq. 1.43.

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \eta \mathbf{R}^{-1} \varepsilon(k) \mathbf{x}(k) \tag{1.44}$$

Note that $\mathbf{x}(k)$ is a vector and $\mathbf{R}^{-1}$ is a matrix, so the update for one weight influences all the other inputs in the system. This is the reason that the computations are no longer local to each weight. However, they are not difficult if one assumes that the inverse of $\mathbf{R}$ is known a priori. The algorithm of Eq. 1.44 is called the LMS/Newton algorithm. The case where $\mathbf{R}^{-1}$ has to be estimated on-line is much more involved and leads to the recursive least squares (RLS) algorithm.

Alternatively, to improve convergence speed with the LMS, we can implement an orthogonalizing transformation of the input correlation function followed by an equalization of the eigenvalues, which is called a whitening transformation (see Appendix A, Section 3.18). Since Newton's method coincides with the steepest descent for performance surfaces that are symmetric, this preprocessing will make the LMS perform as Newton's method.

NEUROSOLUTIONS EXAMPLE 1.20

Newton's method

In this example we implement Newton's method with a custom DLL. For this example we must compute $\mathbf{R}^{-1}$ and apply Eq. 1.44 to the simulator. The autocorrelation function for this example is

$$\mathbf{R} = \frac{1}{15}\begin{bmatrix} 416 & 429 \\ 429 & 490 \end{bmatrix}$$

so $\mathbf{R}^{-1}$ becomes (see Section A.3.11)

$$\mathbf{R}^{-1} = 15\begin{bmatrix} 0.0247 & -0.0217 \\ -0.0217 & 0.0210 \end{bmatrix}$$

By applying Newton's method to the learning algorithm, we have essentially compensated for the eigenvalue spread. This means that Newton's method behaves as the steepest descent for a circular performance surface where the steepest

descent direction always points directly to the optimal value. Thus, although the calculations are more complicated and more demanding (we need to know $\mathbf{R}^{-1}$), the convergence is much faster (in fact, the algorithm can converge in one epoch). When we run the simulator, notice that no matter where the algorithm starts, it always heads directly toward the optimum.

1.9 ANALYTIC VERSUS ITERATIVE SOLUTIONS

Selecting a search procedure to find the optimal weights is a drastic conceptual change from the analytic least square solution, albeit an equivalent procedure. In learning systems the iterative solution is the most common for several reasons.

When working with learning systems, the interest is very often in on-line solutions, that is, solutions that can be implemented sample-by-sample. The analytic solution requires data to be available beforehand to compute the autocorrelation matrix $\mathbf{R}$ and cross-correlation vector $\mathbf{p}$. Fast computers are required to crank out the solution (the inverse of $\mathbf{R}$ and the product with $\mathbf{p}$). The method produces a value that immediately gives the best possible performance. But several problems may surface when applying the analytic approach. If the matrix $\mathbf{R}$ is *ill conditioned,* the computation of $\mathbf{R}^{-1}$ may not be very accurate. Moreover, the analytic solution also requires a great deal of computation time. [Computation of a matrix inverse is proportional to the square of the number of columns D of the matrix. In the big O notation this means $O(D^2)$.]

The iterative solution is not free from shortcomings. We already saw that there is no guarantee that the solution is close to the optimal weight $\mathbf{w}^*$ when all the input samples are used by the algorithm. This depends on the data and on a judicious selection of the step size η. The accuracy of the iterative solution is not directly dependent on the condition number of $\mathbf{R}$, but matrices with large eigenvalue spread produce slow convergence because the gradient-descent adaptation is coupled. As we said previously, the slowest mode controls the speed of adaptation, while the largest step size is constrained by the largest eigenvalue.

The great appeal of the iterative approach to optimization is that *very efficient algorithms exist* to estimate the gradient (e.g., the LMS algorithm). Only two multiplications per weight are necessary, so the computation scales proportionally to the number of weights D [i.e., $O(D)$ time]. Moreover, *the method can be readily extended to nonlinear systems,* while the analytic approach for most cases of practical relevance cannot be computed.

1.10 THE LINEAR REGRESSION MODEL

We started this chapter by pointing out the advantages of building models from experimental data. In the previous sections we developed a set of techniques that adapt the parameters of a linear system (the Adaline) to fit the relationship between the input (x)

and the desired data (d) as well as possible. This is our first model, and it explains the relationship $f(x, d)$ as a hyperplane that minimizes the square distance of the residuals. We will have the opportunity to study other (nonlinear) models in later chapters.

It is instructive to stop and ask the question: How can we use the newly developed regression model? One interesting aspect of model building that we mentioned previously is the ability to *predict* the behavior of the experimental system. Basically, what this means is that once the Adaline is trained, we can forecast the value of d when x is available. We do this by computing the system output y and assume that the error ε is small (Eq. 1.3). You can now understand why we want to minimize the square of the error, since if the square of the error is small, d is going to be close to y in the training data. Figure 1-19 shows a productive way of looking at the input-output pairs that we used to train the Adaline.

We assume that the experimental system produces the desired response d for each input x according to a rule that we do not know. The purpose of building the model is to approximate as well as possible this hidden relationship.

We expect also that even for x values that the system did not use for training, y is going to be close to the corresponding unknown value d. Our intuition tells us that if

- the data used for training covered all the possible cases well,
- we had enough training data, and
- the correlation coefficient is close to 1,

then in fact y should be close to the unknown value d. However, this is an inductive principle, which has no guarantee of being true. The ability to extrapolate the good performance from the training set to the test set is called *generalization.* Generalization is a central issue in the adaptive systems approach since it is the only guarantee that the model will perform well with the future data that will be presented to the system while in operation.

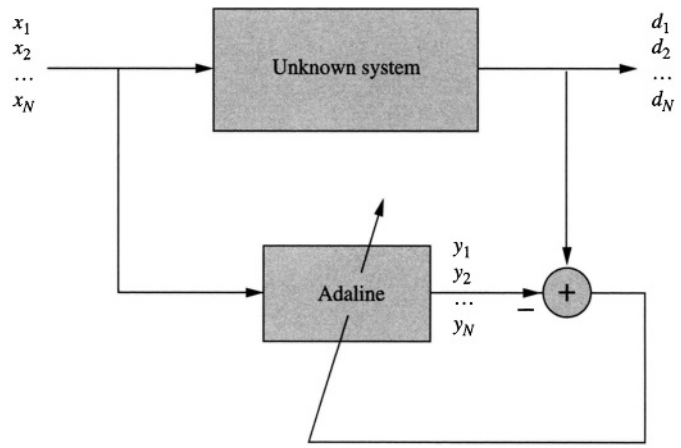


FIGURE 1-19 A view of the desired response as the output of an unknown model

Remember that in the test mode the system parameters must be kept constant; that is, the learning algorithm must be disabled. In the next section we familiarize ourselves with training and using the linear model.

1.10.1 Regression Project

Getting Real-World Data We end Chapter 1 by giving you a flavor of the power of linear regression to solve real-life problems. We will go to the World Wide Web and seek real data sets, import them into NeuroSolutions, and solve regression problems. We will use the Breadboard from Example 1.7.

The first thing is to decide on the data with which we will work. There are many interesting Web sites to visit in the search for data. We suggest the following sites:

climate data: http://ferret.wrc.noaa.gov/fbin/climate_server

http://seamonkey.ed.asu.edu/~behrens/teach/WWW_data.html

These sites have plenty of data (some duplicated). We assume that you know how to connect to the Web and how to download data. You should get the data in ASCII and store it in column format, with one of the variables (the independent variable) in the input file and the dependent variable in the output file. Alternatively, we have provided sample data on the CD-ROM under the Data directory. Read the *readme* file to choose the data sets that interest you.

NeuroSolutions Project The fundamental question is to find out how well a linear relation explains the dependence between the input data and the desired data. We will exemplify the project with a one-dimensional set of input data, but the multidimensional case is similar.

The first thing to do is to modify the NeuroSolutions breadboard so that it will be able to work with the data you downloaded. The data should be stored in an ASCII file and formatted in columns. Right-click on the input file icon and select *properties*. The Inspector will appear on the screen. Remove the present file (click the *remove* button), and click the *add* button. The Windows file inspector will appear; open the file that contains the input data, that is, the input to your linear model.

In NeuroSolutions, the *associate* panel appears, which you can close (we assume that the input file has ASCII data in column format). The next panel that pops open is the *customize* panel. Here you select the columns that you want to use (for those columns that you do not want, select the column label and click the *skip* button), and then click the *close* button. The input file is now open and ready to be used by NeuroSolutions. You should repeat the procedure for the desired data file. Make sure that the number of samples in the input and desired data files are the same.

Another thing that we should do is to normalize the data. Sometimes the input and desired variables have very different ranges, so we should always normalize both the desired and input data files between 0 (or -1) and 1. To do this, go to the *stream* page (click on the *stream* tab) of the Inspector to access the *normalization* panel. Click the *normalize* check box and set the normalization range.

We always recommend that you visually check the data—either with a plotting program or the Scatter Plot in NeuroSolutions—to ensure that there aren't any outliers present in the data. When outliers exist, they may distort any possible linear relationship that may exist.

Once the data sets are open, we can effectively start the adaptation of the linear regressor. The first important consideration is the largest step size that can be used for convergence. When the data is normalized, we can always guess an initial value of 0.1. By plotting the learning curve or the weight tracks (if the problem has few input channels), we can judge how appropriate this value might be. Alternatively, we can compute the eigenvalues and find the exact largest possible step size, but this is rarely done. The trial-and-error method is OK for small problems.

If the problem takes a long time to converge and increasing the step size creates instability, then the eigenvalue spread is large, and there is little we can do short of using Newton's method.

After the algorithm converges (the error stabilizes), we should use the correlation coefficient DLL to estimate the correlation coefficient. Note that it is always possible to pass a hyperplane through some data points, but the real issue is whether the hyperplane provides a good model. To answer this question, we need to estimate the correlation coefficient.

For the multiple-variable regression case, the relative weight magnitude tells us about the relative importance of each variable in the regression equation (for normalized inputs). It is therefore interesting to look at the values of the regression weights, including the bias. Remember that if the data is normalized, the displayed weight values must be "unnormalized" in order to compare them with the original data. You can find the values that NeuroSolutions used to normalize the data by going to the *dataset* page of the Inspector and opening the normalization file. NeuroSolutions multiplies the data set by the first value (range) in the normalization file and adds the second value (offset) in the normalization file. To reverse this process, you must subtract the second value and then divide by the first value.

Remember that the parameters of the regression equation can be used to predict the system output when only the input is known. We can do this by testing the system with another data file for which we do not have a desired response. To do this in NeuroSolutions, you should go to the Controller Inspector (the yellow dial) and turn off the *learning* check box; this fixes the weights.

No problem is finalized without a critical assessment of the results obtained. You should start with a hypothesis about the data relationship and confirm your hypothesis with NeuroSolutions results. If there is a discrepancy between what you expect and the results, you must explain it. This is where the NeuroSolutions probes are very effective. You should verify that the data is being properly read, whether the input and output files are synchronized, whether the system (weight tracks, learning curve) is converging, and so on. Computers are great tools, but they are very susceptible to the "garbage in, garbage out" syndrome so it is the user's responsibility to check the input and the methodology of data analysis.

NEUROSOLUTIONS EXAMPLE 1.21

Linear regression project

We will illustrate the project with a regression between two time series: the sea temperature and atmospheric pressure downloaded from the NOAA site (climate database). We start with the Breadboard from Example 1.7. We replace the input with the file containing the sea temperature and replace the desired response data with the file containing the pressure data. NeuroSolutions automatically sets the number of inputs from the file (verify this in the File Inspector) and the number of exemplars in an epoch. (Verify this in the Controller Inspector). We also have to decide how many iterations we need. In the Controller Inspector enter 1,000 in the *epochs/run* field. This number may be too large, but when the weights and MSE stabilize, we can always interrupt the simulation. Experiment with everything we have learned in this chapter.

1.11 CONCLUSIONS

In this first chapter we introduced ideas that are very important for the rest of the book. Probably the most important was the concept of adaptive systems. Instead of designing the system through specifications, we let the system learn from the input data. To achieve this, the system has to be augmented with an external cost criterion that measures how well the model fits the data, and with an algorithm that will adapt the system parameters so that the minimum of the cost can be reached. We will use this concept throughout this book.

We covered much more in this chapter. We described an extremely simple and elegant algorithm that is able to minimize the external cost function by using local information available to the system parameters. The principle is to search the performance surface in the opposite direction of the gradient. The name of the algorithm is least mean squares (LMS), and in just two multiplications per weight and data sample it is able to move the system parameters towards the neighborhood of the optimal values. Gradient descent is a powerful concept that we will also use throughout this book.

When we apply the LMS to the linear network, we end up with a system called the linear regressor, or Adaline, that can fit hyperplanes to data. The solution is identical to least squares.

We quantified the properties of the LMS algorithm, and we showed the fundamental trade-off of adaptation: the compromise between speed of adaptation and precision in the final solution. We defined the learning curve, which we called the thermometer of learning, that we will also use over and over. Therefore this chapter covers the basic concepts for the intriguing adventure of designing systems that learn directly from data.

We have also provided a project to help you understand the power of adaptive systems. The applications of the Adaline are bounded by our imagination and the data

we can find to train it. Thus, knowing how to get data from the Web and how to use it in NeuroSolutions is of great value.

1.12 EXERCISES

1.1 **(a)** Compute by hand the linear regressor for the following data:

$$X = \{-0.5, -0.2, -0.1, 0.3, 0.4, 0.5, 0.7\}$$
$$D = \{-1, 1, 2, 3.2, 3.5, 5, 6\}$$

(b) Compute the correlation coefficient.

(c) Estimate d for the values $x = 0$ and $x = 1$. Which d can you trust the most? Justify your answer.

(d) Use NeuroSolutions to confirm your results.

1.2 For the data of Problem 1.1, compute the performance surface and plot it. Estimate the gradient at the point $w = 2$.

1.3 **(a)** Compute the linear regression for the following data:

$$X_1 = \{-0.5, -0.2, -0.1, 0.3, 0.4, 0.5, 0.7\}$$
$$X_2 = \{3, 3, 2.5, 2, -1, -1, -4\}$$
$$D = \{-3, -1, 0, 1.2, 1.5, 3, 4\}$$

(b) Use NeuroSolutions to confirm your results.

1.4 Find the eigenvalues and eigenvectors of the following matrix:

$$\mathbf{R} = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix}$$

1.5 Show that when you apply Eq. 1.5 to data that is Gaussian distributed, you effectively obtain a set of linear equations.

1.6 Rerun Example 1.8 with white noise that has a mean value of $m = 0.5$ and explain the results.

1.7 Show mathematically that Eq. 1.14 will always give a value of $|g| < 1$.

1.8 For the data set in Problem 1.3, estimate the largest step size, the critically damped step size, and the time constant of adaptation for each step size.

Run NeuroSolutions and verify your results by plotting the weight tracks. Also verify the time constant of adaptation. For which value does the system adapt the fastest?

1.9 If you know the eigenvalues of the input autocorrelation function, would you schedule the step size for fast adaptation? Justify your answer. Run NeuroSolutions to confirm your conclusion.

1.10 Verify in the data of Problem 1.3 the statement of Section 1.6.5 that when the step size is very large in on-line learning, the MSE in NeuroSolutions can decrease below the theoretical limit. Plot the weight tracks and comment on the validity of the model.

1.11 Compute the largest eigenvalue for the data of Problem 1.3, but now use a regressor with bias. Show all the work. What can you conclude about the largest step size for the bias?

1.12 Apply Newton's search to the data of Problem 1.3 and compare the speed of adaptation with LMS. You should use NeuroSolutions.

1.13 Go to the Data directory and open the folder Auto MPG. Read the instructions and determine the regression equation. Make sure the system is converging. What is the estimated misadjustment for the solution? How would you obtain a better linear fit? Compute also the correlation coefficient and discuss the applicability of your model.

1.14 Repeat Problem 1.13 with the data found in the folder Abalone.

1.15 Construct the NeuroSolutions Breadboard for Problem 1.13 from scratch, that is, one component at a time.

1.13 NEUROSOLUTIONS EXAMPLES

1.1 The linear processing element in NeuroSolutions
1.2 Computing the MSE for the linear PE
1.3 Finding the minimum error by trial and error
1.4 Plotting the performance surface
1.5 Comparison of performance curves for different data sets
1.6 Adapting the linear PE with LMS
1.7 Batch versus on-line adaptation
1.8 Robustness of LMS to noise
1.9 Estimating the correlation coefficient during learning
1.10 The learning curve
1.11 Weight tracks
1.12 Linear regression without bias
1.13 Rattling
1.14 Scheduling of step sizes
1.15 Multivariable regression
1.16 Checking the LMS solution with the optimal weights
1.17 Visualizing the weight tracks and speed of adaptation
1.18 Visualizing weight tracks with on-line learning
1.19 Linear regression without bias
1.20 Newton's method
1.21 Linear regression project

1.14 CONCEPT MAP FOR CHAPTER 1

REFERENCES

Casti, J. L., *Alternate Realities: Mathematical Models of Nature and Man,* Wiley, 1989.

Dunteman, G., *Introduction to Linear Models,* Sage Publications, 1984.

Haykin, *Adaptive Filter Theory,* Prentice Hall, 1996.

Melsa, J., and D. Cohn, *Decision and Estimation Theory,* McGraw-Hill, 1978.

Widrow, B., and S. Stearns, *Adaptive Signal Processing* (Chapter 4), Prentice Hall, 1985.