Theory and Application of Backpropagation: A Handbook

Paul W. Munro

dedicated to the memory of Frank Rosenblatt (1928-1971), who was on the right track.

"*In questions of science, the authority of a thousand is not worth the humble reasoning of a single individual*"

- Galileo Galilei

Preface

"What is a neural network, anyway?"

Occasionally confronted with such a question (by brave souls in social settings persisting beyond the requisite, "What sort of work do you do?"), I find myself describing an approach to computer simulation of neural function. This often leads to a speculative discussion of the limits of machine intelligence and whether robots will someday rule the world. Assuming a more focused audience here, I submit the following response:

> The term *neural network* often brings to mind thoughts of biological devices (brains), electronic devices, or perhaps network diagrams. But the best conceptualization, at least for the discussion here, of a neural network is none of these; rather, think of a neural network as a mathematical function. Not only does the abstractness of mathematics allow a very general description that can unite different instantiations of neural information processing principles, it is also inherently subject to formal analysis and computer simulation.

Borrowing the dynamical systems approach of the so-called hard sciences, the field of neural networks has progressed in numerous directions, both towards developing explanations of the relationship between neurobiology and behavior/cognition and towards developing new approaches to problems in engineering and computer science. The field has surprisingly deep roots (see Anderson's and Arbib's books for references), and arose in this century with the promising work of Hebb (1949), McCulloch & Pitts (1943), and Rosenblatt (1962). Progress slowed to a near halt from the mid 60s to the late 70s due, in large part, to a vicious debate between two camps in the fledgling AI community. One camp asserted that AI should be grounded in symbols and logic. The other camp sought a basis grounded in neurobiology.[1] One of the over-hyped promises of the latter approach was that research in neural networks would lead to robust techniques for training by example; e.g ., given a set of images of cats and a set of images of dogs, such a technique might be able to extract, directly from the data, the distinguishing features, so that a novel stimulus could be correctly classified as a cat or a dog. In his book, *Principles of Statistical Neurodynamics*, Rosenblatt (1962) outlines several approaches to developing such a system, but was never quite able to solve the puzzle.

---

[1]A study of the history and the politics surrounding this AI debate would make a fascinating book. This is not that book however; for a historical perspective, the reader is referred to Crevier's 1993 book.

The failure to produce such a technique, and the proof by Minsky & Papert (1968) of the ultimate futility of Rosenblatt's line of research, led to the abandonment of research in neural networks by the computer science community. Certain physicists and mathematicians (Little & Shaw, 1978; Nass & Cooper, 1975; Hopfield, 1982; Grossberg, 1976; Fukushima, Miyake, Ito, 1983) began to reconsider neural networks as information processing models. These were followed by some mathematical psychologists (Anderson et al., 1977; McClelland and Rumelhart, 1981) interested in using similar models to account for cognitive phenomena. The research money that was closed to neural network research in the 1960s began to trickle back in the late 1970s. Supported in large part by grants from such agencies as the Sloan Foundation and the Office of Naval Research, a few research centers arose that brought together interested scientists from a diverse range of disciplines. Neurobiologists, psychologists, physicists (and even some renegade computer scientists!) have attempted to develop a new framework that might lead to the unification of neurobiology and cognitive psychology. Ultimately, it became clear that a slight modification to Rosenblatt's assumptions enabled the solution that he had sought. This discovery generated a degree of excitement and interest that is rare in science. This finding, commonly referred to as the *backpropagation learning procedure,* has been responsible more than any other for the tremendous growth in neural network research over the past decade. The goal of this book is to explain backpropagation, from its foundation through its derivation and how it is applied as a tool for both understanding cognitive processes and developing novel approaches to information processing tasks.

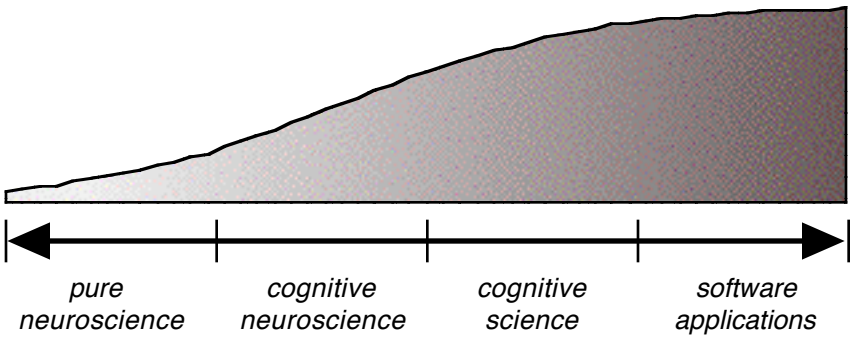


Figure 0.1. The author's conception of backpropagation's influence across a spectrum of neural network research with respect to neurobiological plausibility.

In spite of their neurobiologically inspired roots, neural network based models are often dismissed, ignored, or ridiculed by neurobiologists for their lack of neuroscientific relevance or plausibility. Within the neural network research community, there is a broad spectrum of views as to what degree of contact with neuroscience is appropriate. At one extreme lie computer models intended to describe neurobiological systems as accurately as possible; at the opposite end is software that uses neural network principles as an approach to tasks where traditional programming principles have failed. In the middle of the spectrum lie neural network models of cognitive processes that vary over virtually the entire range of neurobiological plausibility. Even though the backpropagation procedure

(henceforth, *backprop*) is a central theme in the neural network community (indeed, it is virtually synonymous with neural networks for many), it violates several key principles of neuroscience. Backprop is found much more frequently on the applied end of neural networks research than on the neuroscience end, and has made a significant contribution toward cognitive science (see Figure 0.1).

Throughout this spectrum, there are two computational properties of neural network models that have generated interest:

- Computation that emerges from the cooperative interaction of many units
- Networks that learn by example and generalize to novel stimuli

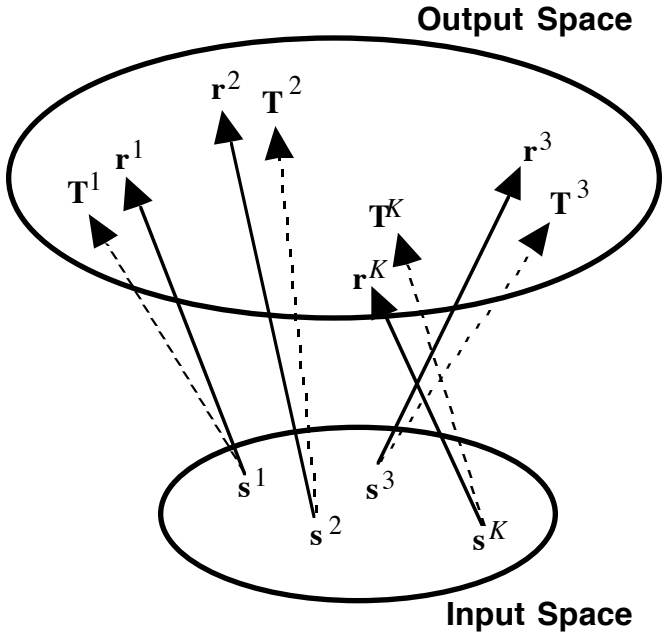


Figure 0.2. A system mapping and an assigned task. The stimuli are mapped from the input space to responses in the output space by the system (solid arrows). A task is defined as a set of **s**-**T** pairs (dashed arrows).

For the most part, although not entirely, models tend to exhibit one of these properties, but not both. This roughly mutual exclusivity reflects conflicting constraints on the architectures of the networks. Cooperative interaction generally requires reciprocal connections between units in a network (if unit *A* excites unit *B*, then unit *B* excites unit *A*). Unfortunately, the existence of such connections often makes neural network learning procedures much more complicated (and slower). While the reasons are somewhat technical and beyond the scope of this preface, the reader will hopefully be satisfied for the moment. This book deals with the latter of the bulleted properties, learning and generalization. Here, the property of *generalization* is defined as the capacity of a system to be "instructed" from a "training sample" of stimulus-response pairs, to somehow "extract" a rule or relationship

between these responses and the corresponding stimuli. A system that generalizes well can generate appropriate responses to stimuli that were not included in the training sample. Assuming a systematic relationship exists between stimulus and response[2], we seek a technique for discovering it from examples. As we will see later (Chapter 5), the degree of generalization is measurable in an approximate sense.

Consider a learning system **L**, which generates a response **r** as a function of its input (stimulus) pattern **s** and a set of internal parameters $\{p_1,...p_N\}$. A set of K stimulus patterns $\{\mathbf{s}^1,...,\mathbf{s}^K\}$ is thus mapped to a corresponding set of responses $\{\mathbf{r}^1,...,\mathbf{r}^K\}$, where the mapping depends on the values of the parameters. The system **L** is typically assigned a *task*, which is defined in terms of a *desired mapping*. The desired mapping is described, either partially or entirely, in terms of a *training set* of stimulus-target pairs of the form $\{(\mathbf{s}^1,\mathbf{T}^1),...,(\mathbf{s}^K,\mathbf{T}^K)\}$. The objective of a learning system is to find the set of parameters that gives a mapping as close as possible to the task. Bear in mind that the space of possible mappings is limited by the functional definition of **L**, and thus there may not exist a set of parameters that will give an exact match between the mapping and the given task. In fact, as we will see, an *exact* match may not be desirable even when it is possible.

In order to compare the performance of one set of parameters to another, some measure of performance is required. This is usually based on the differences between the responses and targets for each stimulus pattern. Once a performance measure has been defined, there are several approaches to finding a set of parameters that optimize it:

Direct computation: For some types of functions, the optimal parameters can be directly computed from the training set. Linear regression (discussed in Chapter 3) is an example of this kind of method.

Random search: Random sets of parameters are generated and tested (this method is also known as "generate and test"). In order to test a set of parameters, the responses of **L** to all the stimuli and the corresponding targets must be compared.

Directed Search: The system begins with a random set of parameters that are modified incrementally such that the responses "improve" (approach the targets) over time.

Any process used to find the parameters can generally be called *learning*, but for the purpose of this volume, we will consider learning to be a gradual process (like directed search), whereby the response properties (behavior) of a naive system are progressively changed.

Several people deserve credit for the realization of this book. The initial concept was encouraged by Betty Stanton of MIT Press. Both she and her late husband Harry were instrumental in developing this project, as they have been

---

[2] If the responses in the training set are arbitrary with respect to the training stimuli (i.e. there is no systematic relationship), generalization is not possible.

in the publication of many seminal publications in the neural networks area (among other areas within the cognitive and neural sciences). Who knows where neural networks would be without the involvement of publishers with their vision? Other individuals at MIT have also been great assets including Teri Mendelsson and more recently Doug Sery. Jim Anderson's influence has also been of critical importance, both in support of this project from its inception, and for introducing me to neural network models of cognition in his *Understanding the Brain* course back in the late 70s. The comments of Richard Golden and several anonymous reviewers were extremely valuable. The School of Information Sciences at the University of Pittsburgh has been generous in supporting the time I have spent on this project as well as providing computational resources and technical help necessary to the publication of the book. This includes the assistance of some graduate students, most prominently Bambang Parmanto, Thea Ghiselli-Crippa, Denis Nkweteyim, and Sejo Pan. A great deal of the writing was accomplished in a number of the coffee shops that fortuitously appeared in Pittsburgh just about the time I began the book. Although he was not directly involved with the production of this particular book, I would like to thank Dave Rumelhart for first explaining the backpropagation procedure to me, when I was very fortunate to have worked as a postdoctoral fellow at UCSD in the mid-80s. Likewise, I would like to express my thanks to the very many researchers who have participated in the development of the procedure. This would include just about everyone referenced in the bibliography and more, but because of their broad influence, certain individuals stand out whom I would like to explicitly recognize: Jeff Elman, Geoff Hinton, Yann LeCun, Jay McClelland, Terry Sejnowski, and Paul Werbos. Love and gratitude go to my friends for their patient support. And of course, my greatest debt of gratitude goes to Avi Baran Munro, my life partner, for providing emotional support throughout this long process, and for putting up with the late or missed meals, not to mention valuable proofreading. I must also thank my four children for adding so much to my life, providing distraction from my book (mostly welcome distractions), and for ensuring that the book progressed at a most cautious and deliberate pace.

This volume is intended to fulfill two roles; it is written both as a textbook and as a reference resource. As a textbook, I have opted for a tutorial form with many examples and exercises. Any student with an understanding of calculus should be able to follow the material, and the methods are illustrated with PASCAL code to clarify the equations. The book could form the basis of a course for upper level undergraduates majoring in just about any quantitative field. However, as a subject for a semester-long course, backprop will certainly strike many professors knowledgeable in the field as a bit too narrow. Thus, I anticipate that it will find use as a secondary text in some neural network classes. The reference function of the book is fulfilled primarily in Parts III and IV, which include descriptions of many extensions to, and applications of, backprop. Since the body of relevant literature is vast (literally in the thousands of papers), a carefully selected sample of classic papers and personal favorites are described here. I hope that many readers who are introduced to this book in a course will continue to find it useful in their personal libraries.

The chapters are organized into four parts, with nicknames stemming from the use of the term "vanilla backprop" by the neural network community to refer to basic unadorned backpropagation:

- **Part I ("Cream and Sugar")** presents a set of concepts from neural networks, numerical methods, and statistics. These "ingredients" provide background for the introduction of the backprop procedure.
- **Part II ("Vanilla")** combines the ingredients from Part I to derive basic backprop, presents some basic examples, and discusses some general practical considerations.
- **Part III ("Tutti Frutti")** is a collection of techniques for enhancing the basic algorithm that have greatly extended its applicability. A variety of flavors of backprop are described.
- **Part IV ("Banana Split")** includes descriptions of several applications of backprop in neurobiology, cognitive science, and statistical analysis of data.

Part I. Background: *Cream and Sugar*

"Its realm is accordingly defined as that part of the total sum of our knowledge which is capable of being described in mathematical terms." -- Albert Einstein (1950) in reference to the realm of Physics

This first set of chapters introduces a set of principles that can be used to derive and analyze the backpropagation procedure which is presented in its pure form (often called "vanilla" backprop in the literature) in Part II. Here, the ingredients to the theoretical development are:

- computation by neural elements (called units), individually, and in networks
- determination of a system's parameters by statistical regression
- dynamical systems and gradient descent

The reader should be aware that these principles are not *necessary* to the development of the theory, but they are *sufficient*; that is, there are alternative approaches to the theory of backpropagation. The emphasis on dynamical systems in this approach comes from the domain of classical physics (a la Newton). Thus, readers with an understanding of elementary calculus should be able to grasp the theory (some knowledge of differential equations would help). A passing familiarity with college level mathematics and an understanding of programming should give one enough preparation to explore neural network simulation and backpropagation.

1. The Single Unit

Neural networks are constructed from computational elements that will be called units in this book; other sources use a variety of names: for example, neurons, nodes, NCEs (neural computational elements). This section is a discussion of the properties of single units (in isolation from a larger network). A single unit can be considered the simplest neural network architecture (much as a single atom is the simplest molecular structure).

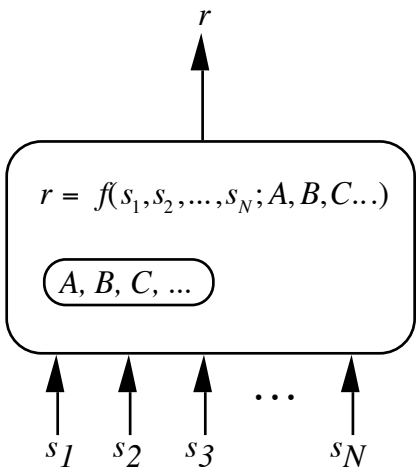


Figure 1.1. A general unit. The response of the unit is computed as a parametric function of the stimulus components s_1, s_2, ..., s_N. The parameters (A, B, C, ...) can be used to alter the functional relationship between stimulus and response.

A unit generates an output (r) in response to some set of input stimuli (s_i, i=1...N). Since we are only interested in the computational properties of the unit, it is completely characterized by its *transfer function* (i.e., its input/output function) f. As we will soon see, the transfer function is usually expressed parametrically; that is, the response is determined not only by the inputs s_i, but also by some internal parameters (for the moment, let these parameters be

denoted *A*, *B*, *C*...) that generally vary from unit to unit within a network. The remainder of the chapter explores some particular classes of transfer functions. As we will see, the parameters are essential for specifying learning rules.

1.1. Linear Units

Mathematical definition of a linear unit

A good place to begin is with the example of a transfer function that is simply a weighted sum of the stimulus components. Using vector notation, the response is conveniently expressible as the inner product (also known as the dot product) by defining a stimulus vector $\mathbf{s} = (s_1, s_2, s_3, \ldots, s_N)$ and a weight vector, $\mathbf{w} = (w_1, w_2, w_3, \ldots, w_N)$. The weights are a special case of the parameters *A*, *B*, *C*, etc. from Figure 1.1. For now, let the term *Fanin* refer to the number of inputs (in later chapters, the term will be refined to connote the sources of the inputs rather than just their number).

The linear transfer function:

$$r = w_1 s_1 + w_2 s_2 + \cdots + w_N s_N = \sum_{i=1}^{N} w_i s_i \qquad [1\text{-}1]$$

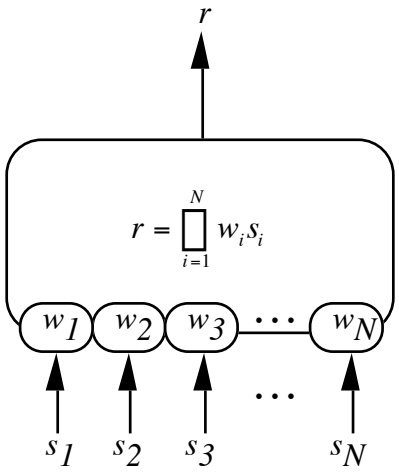


Figure 1.2. A linear unit. The response *r* of a linear unit is the sum of the products of the *N* stimulus components, s_i, each multiplied by a corresponding "synaptic weight", w_i.

The PASCAL function below computes the output for a linear unit:

```
function linxfer(stim: Array[1..maxin] of Real,   {Stimulus pattern}
                 wts: Array[1..maxin] of Real,    {Weight vector}
                 fanin: Integer)                  {The number of inputs}

var ii : Integer ;
    x : Real :
begin
    x := 0.0 ;
    for ii := 1 to fanin do begin
       x := x + wts[ii]*stim[ii]
       end ;
    linxfer := x
end;
```

If a particular set of weights is chosen, then a numerical value can be computed for the response generated by any stimulus vector. Thus, a unit reduces a pattern of input values (a vector quantity) to a single output value (a scalar quantity). It is important to understand that *different* input vectors can give the *same* response (see the example below).

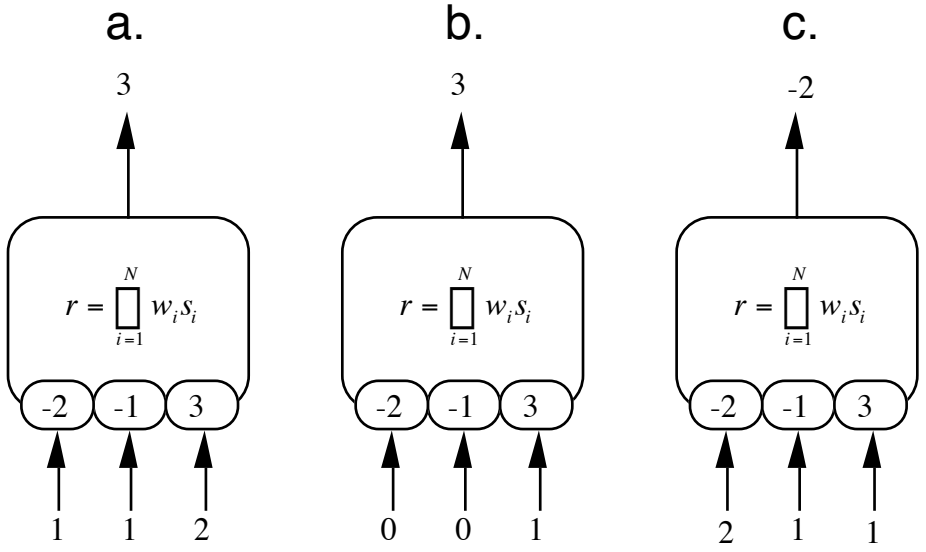
Example 1.1 Linear computation

For the weight vector w = (-2, -1, 3), compute the responses to the stimulus vectors.

a. s = (1,1,2) Solution: $r = w_1s_1 + w_2s_2 + w_3s_3 = (-2)(1) + (-1)(1) + (3)(2) = 3$

b. s = (0,0,1) Solution: $r = w_1s_1 + w_2s_2 + w_3s_3 = (-2)(2) + (-1)(0) + (3)(1) = 3$

c. s = (2,1,1) Solution: $r = w_1s_1 + w_2s_2 + w_3s_3 = (-2)(2) + (-1)(1) + (3)(1) = -2$



Contours of constant response

Note that both the weight vector and the stimulus vector have the same number of components (*N*). In this section, the response of a unit will be plotted over an *N*-dimensional coordinate system, where *N* is the number of stimulus components (and for plotting purposes, *N* is generally equal to 2). There are generally two complementary perspectives for generating these plots:

- Stimulus space: Each point in the space corresponds to a stimulus vector. In stimulus space, a particular weight vector **w** induces a "response field"; that is, each point **s** in the space is assigned a value equal to the response of a linear unit, **w•s**.
- Weight space: Each point in the space corresponds to a weight vector. In weight space, a particular stimulus vector **s** induces a "response field"; that is, each point **w** in the space is assigned a value equal to the response of a linear unit, **w•s**.

Contours in stimulus space

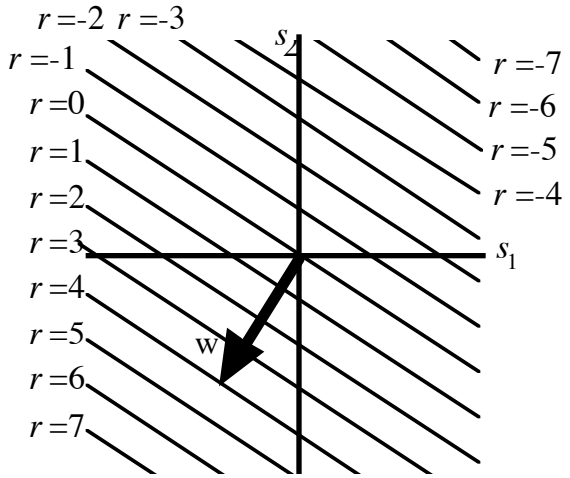


Figure 1.3. Isoresponse contours for a linear unit with two inputs. The stimulus vectors along any line orthogonal to the weight vector generate a constant response. See text for explanation.

We have seen that a unit responds with a single value to a multi-valued stimulus. Imagine an *N*-dimensional coordinate system, in which the axes represent the stimulus components, so each point in the space corresponds to a hypothetical stimulus pattern; that is, each stimulus pattern can be conceptualized as a point in *N*-space. If the transfer function (including parameters) for a given unit is specified, then the numeric value of the response to any stimulus pattern can be computed; thus a given unit specifies a "field" of response values on the stimulus space.

For a two-dimensional stimulus space, the structure of the field can be illustrated by contours of constant response, these *isoresponse contours* are like isotherms on a weather map, or isobars on a topographic map. For a linear transfer function, these contours are parallel lines; the weight vector (which can be represented in the same space since it has the same number of dimensions) is in the perpendicular direction. This is illustrated in Figure 1.3 for a linear unit with two inputs having weight values -1 and -2.

For a weight vector with three components, the contours of constant response are planes (as in Figure 1.4), all perpendicular to the weight vector (and thus parallel to one another).

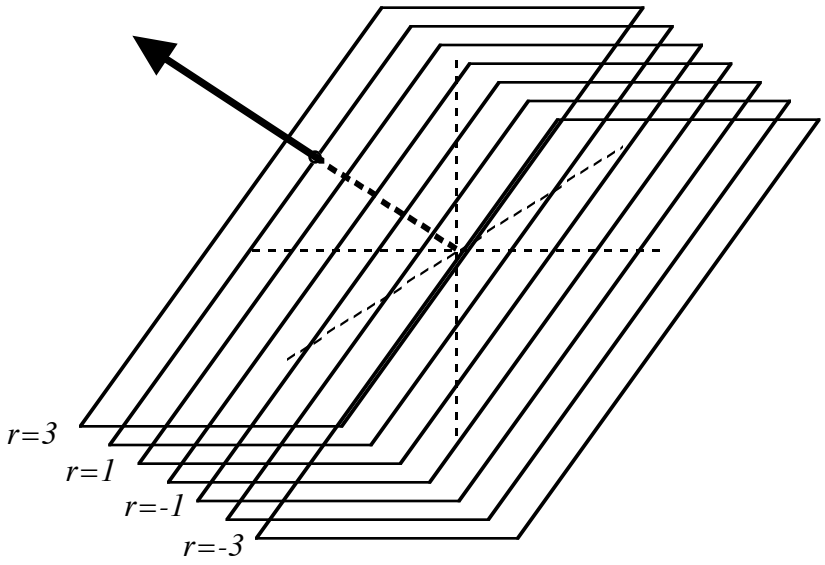


Figure 1.4. Isoresponse contours in a stimulus space of three dimensions. See text for explanation.

Contours in weight space

The linear response function is symmetric with respect to the weight and stimulus vectors; that is, if the two vectors were exchanged, the response would not be affected (see the example in Figure 1.5). This can be demonstrated easily using the procedure **`linxfer`**.

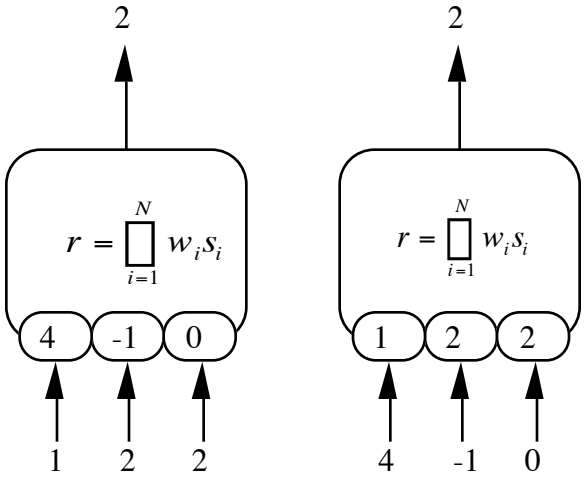


Figure 1.5. Symmetry between weight vectors and stimulus vectors. An example illustrating the symmetry between a weight vector and a stimulus vector. Note that interchanging the stimulus and weight components does not alter the response value.

Just as a weight vector can induce a response field on stimulus space, a stimulus vector can induce a response field on weight space. Because of the weight/stimulus symmetry of the linear transfer function, the isoresponse contours in stimulus space are linear (lines in 2D, planes in 3D, etc.). They have the same property in weight space. Hence the figures depicting weight vectors in stimulus space (Figures 1.3 and 1.4) would apply if the axes were labeled w_i, and the vector was labeled **s**.

Two inputs (2D stimulus space and weight space)

For example consider the task specified by:

Stimulus	Response
6 −2	14
−2 4	2

The two items in this task put *constraints* on the allowed weights; i.e. the weight values that will yield the desired response to each corresponding stimulus pattern. This can be seen by considering the space of all possible weights (in 2D, the weight space is a plane). The first item on the task list states that if the first stimulus value is 3 and the second is -1, the response should be 7. Since there is an isoresponse contour in weight space corresponding to each possible response value for a given stimulus vector, this item corresponds to a contour (for a linear unit, it is a line with an orientation perpendicular to the stimulus vector). To plot the precise contour for a particular item, insert the stimulus and response values into the expression for a linear unit, and graph it in the w_1-w_2 plane:

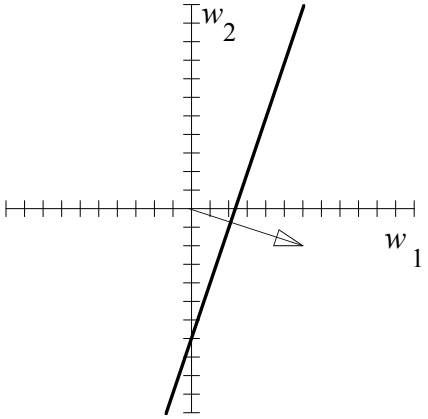


Figure 1.6. An isoresponse contour in weight space. The contour (bold line) corresponds to the set of weight states that result in a response $r = 14$ to the stimulus $s_1 = 6$, $s_2 = -2$.

The second task item (r=2 when s_1 =-2, s_2 =4) imposes a different constraint on weight space, and the set of weights satisfying both constraints is defined by the intersection of the two constraints. In this example, the intersection is a single point since the constraints are lines; however, this is not always the case, as will be illustrated in the next section.

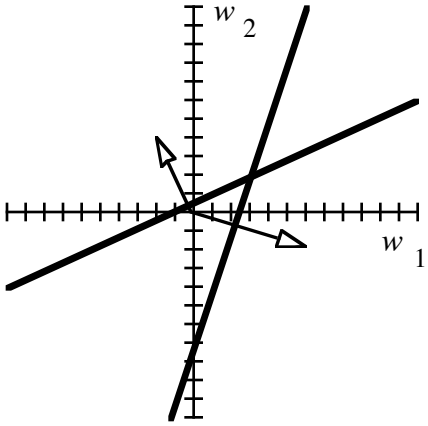


Figure 1.7. Task items impose multiple constraints on weight space. Weights that simultaneously satisfy both constraints lie at the intersection. For the constraints in the example, the two items are satisfied only by a single weight vector: (3,2).

More than two inputs

For a linear unit with more than two inputs, each item constrains the allowed weights to a subspace with a dimensionality that is one less than the original weight space. For example, each stimulus-response item for a unit with

three weights (recall Figure 1.4) constrains the weights to a two-dimensional plane perpendicular to the stimulus vector. A task consisting of two such constraints would intersect along a line (see Figure 1.8); i.e. all weights that simultaneously satisfy both constraints must lie on this line. Note that the addition of a third constraint will generally restrict the solution space to a single point in weight space.

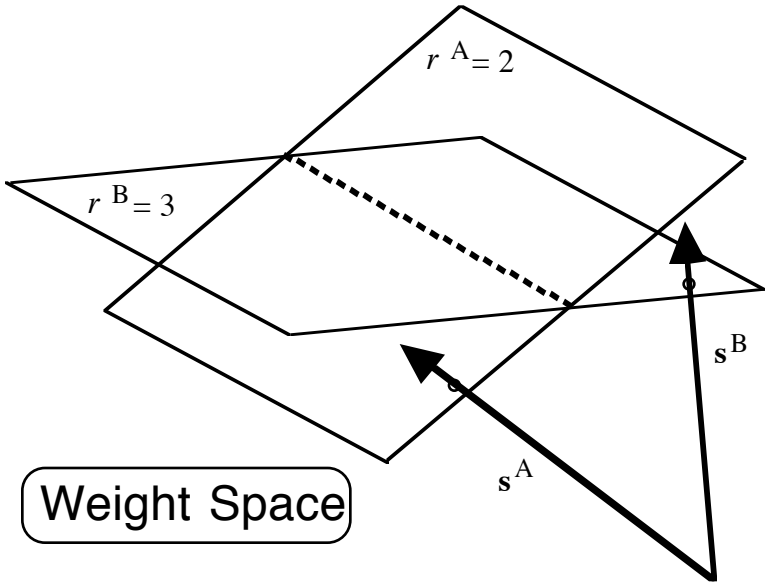


Figure 1.8. Intersecting isoresponse contours in 3 dimensions. Isoresponse planes are shown for two stimulus vectors in weight space: the response r^A to stimulus $\mathbf{s}^A$ is 2 and the response to $\mathbf{s}^B$ is 3. The planes are orthogonal to the corresponding stimulus vectors. Weights that lie on the intersection of the two planes (dashed line) simultaneously satisfy both constraints.

1.2. Linear Threshold Units

Mathematical definition

The mathematical properties of the linear transfer function are extremely well understood and subject to a rich set of formal analytical techniques. Unfortunately, this function has an intrinsic property that severely limits its usefulness. This fatal flaw will be described later at the beginning of the discussion of multilevel networks. Fortunately however, some aspects of the linear units can be retained that will make use of the visual intuitions developed in the previous section. A linear threshold transfer unit with N inputs $(s_1, ...,s_N)$ has $N+1$ parameters $(w_1,...,w_N,\theta)$. The function is defined by first computing the same weighted sum as before:

$$x = w_1 s_1 + w_2 s_2 + \cdots + w_N s_N = \sum_{i=1}^{N} w_i s_i = \mathbf{w} \bullet \mathbf{s}$$

The response is then one of two values, a or b, depending on whether x is greater than or less than the threshold parameter θ. This "step function" is a common technique for mapping variables from a continuous domain to variables on a discrete range.

$$r = \begin{cases} a & \text{if } x < \theta \\ b & \text{if } x \geq \theta \end{cases}$$

Example 1.2 A linear unit with three inputs and two patterns.

Consider the task in the table.

Stimulus	Response
1 2 −1	4
3 −2 0	1

The two constraints imposed by these items are expressed by the equations,

$$w_1 + 2w_2 - w_3 = 4$$

$$3w_1 - 2w_2 = 1$$

Since the intersection of the constraints is a line, there are an infinite number of weight states that solve the task: One solution to these equations is **w** = (1,1,-1); another is **w** = (3,4,7). Plug in any value for one of the **w** components, and the equations can be solved for the other two. The intersecting line can thus be expressed in terms of a single constant (k). By setting w_1 to a specific value (k), and solving the equations for w_2 and w_3, all vectors which satisfy both of these constraints can be expressed in the form:

$$\mathbf{w} = \left\{ k, \frac{3k-1}{2}, 4k-5 \right\}$$

Different values of k specify different points along the line. [This is an example of a one-dimensional manifold; a subspace within which a point can be specified by m parameters is called an *m-dimensional manifold*.]

The PASCAL function below computes the output for a linear threshold unit:

```
function thrxfer(stim:Array[1..maxin] of Real,    {Stimulus pattern}
                 wts: Array[1..maxin] of Real,    {Weight vector}
                 thresh: Real ;                   {Threshold}
                 fanin:Integer)                   {The number of inputs}

var ii : Integer ;
    x  : Real :
begin
    x := 0.0 ;
    for ii := 1 to fanin do begin
```

```
        x := x + wts[ii]*stim[ii]
        end ;
    if x < thresh then thrxfer := a ;
                  else thrxfer := b ;
end;
```

Usually, the values of *a* and *b* are 0 and 1 respectively, but often -1 and +1 are used. In principle however, any two values could be used (see Figure 1.9).

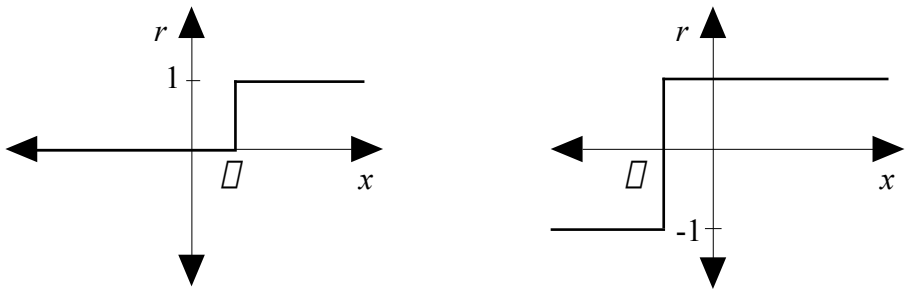


Figure 1.9. The threshold function. Two examples of the threshold function are shown for commonly chosen values of *a*, *b*, and . **Left:** *a*=1, *b*=0, >0. **Right:** *a*=1, *b*=-1, <0.

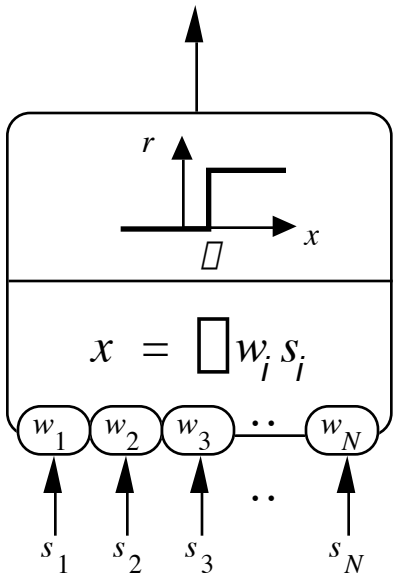


Figure 1.10. A linear threshold unit. The weighted sum (*x*) is computed and passed through the unit's threshold function. The unit has *N* weight parameters and a threshold parameter.

Regions of constant response in stimulus space

Since an LTU has only two possible output values, the response field over the stimulus space is characterized by large *regions* of constant response (rather than contours), where each region can be labeled by the value of the response (*a* or *b*). Consider the response field of a linear unit induced by a given weight vector in stimulus space (as

in Figure 1.3 earlier). In the context of an LTU, the lines that were contours of constant response are now contours of constant x. Note that x increases continuously in the direction of the weight vector. Thus, there is a linear contour (a line in a 2-D space, a plane in 3-D, etc.) along which $x=\theta$. This contour (the bold line in Figure 1.11) divides the stimulus space into two half-spaces, one having $x>\theta$, and hence $r=a$ in this region, whereas in the other half-space, $x<\theta$ (i.e. $r=b$).

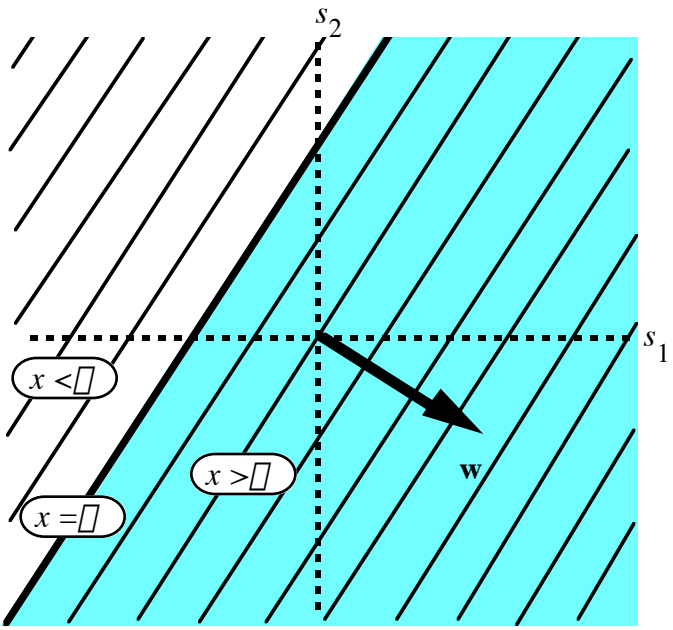


Figure 1.11. Separation of a space by an LTU creates a linear boundary. The response field induced by an LTU with a given weight vector (**w**) and threshold (θ) on a stimulus space. The weighted sum x is constant along each of the parallel lines running perpendicular to **w**. One of these (the bold one) corresponds to $x=\theta$. On the shaded side of this line in the direction of **w**, $x>\theta$, and hence $r=a$; on the unshaded side, $r=b$ (since $x<\theta$).

Since an LTU divides the stimulus space into two regions with a linear boundary, it is only capable of solving tasks that can be so divided; these tasks are termed *linearly separable*. Some examples of a linearly separable (LS) task and a non-LS task are shown in Figures 1.12 and 1.13 for a 2-D stimulus space.

Stimulus		Response
-9	3	1
-7	1	1
-3	-3	1
7	2	0
11	11	0
13	-5	0

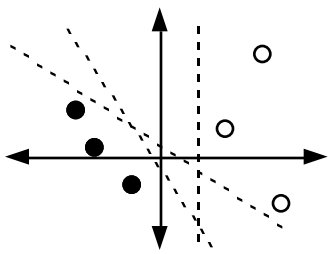


Figure 1.12 A linearly separable task. Note that tasks defined by a finite set of data points generally have an infinite number of linear separating boundaries, three of which are shown (dashed lines).

If a set of stimulus-response items is not LS, they may be easily separated by some other type of simple contour (see Figure 1.13 for example).

Stimulus	Response
-3 2	1
3 -2	1
3 5	1
5 11	0
7 -8	0
-9 2	0

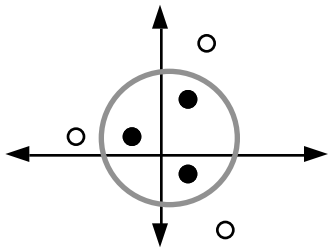


Figure 1.13. A non-LS task in 2-D. The two classes cannot be separated by a single straight line. Note however, that the classes may be easily separated by some other type of simple unit response function (such as one that specifies a circular boundary, in this case).

Given an LS task, there exists a set of LTU parameters (weights and threshold) that can solve it. If an appropriate weight vector can be found, a threshold value can be determined.

Stimulus	Response
-5 3	0
-1 -1	0
1 3	1
5 -1	1

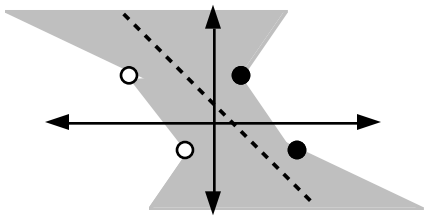


Figure 1.14 Nonuniqueness of separating boundary between finite data sets. An LS example with a stimulus space plot showing stimuli generating r=1 shown with filled circles, and non-responsive (r=0) stimuli represented by open circles. Any straight boundary in the shaded region can separate the two classes.

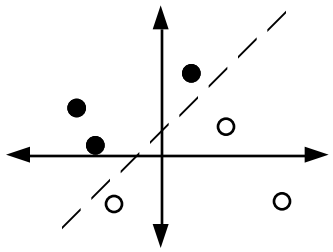
Even without plotting the points, a solution is evident: all the stimuli generating a response of zero have a negative value for the first stimulus coordinate, and the stimuli generating a response of one have a positive value. This observation can be expressed in the form of a linear threshold function: if $s_1>0$, then r=1; otherwise r=0. This is equivalent to setting w_1=1, w_2=0, and θ=0. Note that the resulting discrimination line is the y-axis (perpendicular to the weight vector [1,0]).

As was mentioned before, there are other solutions as well. Consider one such alternative discrimination boundary running diagonally from the upper left to the lower right (the dashed line). To find a set of weights and thresholds corresponding to a diagonal line, first choose a weight vector perpendicular to the desired boundary. If the weights have been chosen correctly (and this is a big "if"), then finding an appropriate threshold is a trivial exercise. To

determine whether the weight vector is appropriate, simply compute the linear sums, and determine the maximum linear sum among the patterns in the below-threshold class $x^b_{\max}$ and the minimum linear sum in the above-threshold class $x^a_{\min}$. If $x^a_{\min} > x^b_{\max}$, then the weight vector can separate the classes with a threshold that lies between $x^b_{\max}$ and $x^a_{\min}$.

Example 1.3. Find a set of weights and thresholds for an LTU that will generate the task listed below.

Stimulus		Response
-9	3	1
-7	1	1
3	9	1
7	2	0
-5	-5	0
13	-5	0



The first step is to decide whether the task is linearly separable, and if so, identify the orientation of a separating boundary. The weight vector is perpendicular to the boundary. If a weight vector exists that generates linear sums for the "1" stimuli that are all greater than all the linear sums for the "0" category, then the task is linearly separable. Once such a weight vector can be determined, finding a threshold value is relatively straightforward. A useful first step for a task in two dimensions, such as this one, is to plot the points (shown above). In this example, it is evident from the graph that a separating boundary can be drawn at a 45^{O} angle. Two weight vectors are possible: (1,-1) and (-1,1). Both of these vectors will separate the stimuli; i.e. either vector when applied to the stimulus set will give linear sums that can be separated by a threshold value. For one of them, all the "0" stimuli will have sums greater than the "1" stimuli, and for the other, the reverse will hold. The desired vector is the latter, and can be determined either by calculating the linear sums and checking, or by visual inspection: *The correct weight vector points in the direction of* ***higher*** *linear sums; i.e. toward the* "1" *region.* This is the vector (-1,1). To find an appropriate threshold value, compute the linear sums (as shown in the table).

Stimulus		Weighted Sum	Response
-9	3	12	1
-7	1	8	1
-7	9	16	1
3	2	-1	0
-5	-5	0	0
13	-5	-18	0

Since the lowest weighted sum in the "1" class ($x^{a}_{\min} = 4$) is greater than the largest sum in the "0" class ($x^{b}_{\max} = 0$), a threshold θ can be found that satisfies $x^{a}_{\min} > \theta > x^{b}_{\max}$; thus any value between 0 and 4 can

The magnitude of the vector is irrelevant, but the *direction* is critical. Let *a* and *b* be set to 1 and 0 respectively. Consider the following weight vectors, which are all perpendicular to the desired boundary: (-2,-2), (1,1), (3,3), and (6,6). It is important to select a vector that *points toward the side of the boundary with a desired response of 1* (*a*). The table shows the resulting linear sums (before threshold). Note that for the first three weight vectors, the condition $x^{a}_{\min} > x^{b}_{\max}$ is satisfied, and a threshold can be found (θ=0 works in all three cases), however the fourth

weight vector, (-2,-2), cannot give a solution for the task since the condition is not satisfied. Note that reversing the inequalities in our threshold function definition would permit (-2, -2) to solve the task, since the new condition would be reversed as well: $x^a_{\min} < x^b_{\max}$.

Stimulus Vectors	Target Responses	Weight Vectors			
		(1,1)	(3,3)	(6,6)	(-2,-2)
-5 3	**0**	-2	-6	-12	4
-1 -1	**0**	-2	-6	-12	4
1 3	**1**	4	12	24	-8
5 -1	**1**	4	12	24	-8

Regions of constant response in weight space

Just as a given weight vector partitions stimulus space into regions that yield "true" and "false" with a straight line, a given stimulus vector partitions weight space with a linear boundary (for example, by reversing the "*s*" and "**w**" labels in Figure 1.11). For a linear threshold unit, any particular stimulus vector **s** induces a linear boundary in the perpendicular direction that divides the weight space into two half-spaces. For a given value of the threshold, each point in the weight space represents a unit with the corresponding weights. Thus, units with weights on one side of the boundary give a response of *a* to **s**, while points on the other side yield a *b*. A set of stimuli break the space into regions that can be labeled according to a *pattern* of responses. That is, any particular set of weights and threshold can be characterized by the responses it generates across some standard set of stimuli. An example is shown in Figure 1.15 for two stimuli. Assuming a threshold value θ=1, a region in weight space can be identified for which the response to both test patterns (here labeled **u** and **v**) is zero. Another region corresponds to responses to **u** and **v** of **0** and **1** respectively, and correspondingly for (**u**,**v**)=(**1**,**0**) and (**u**,**v**)=(**1**,**1**).

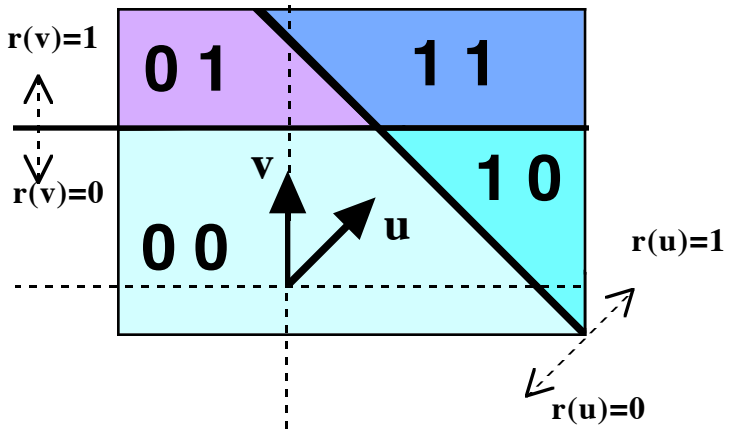


Figure 1.15 Partitioning of weight space into four regions by two stimulus vectors. The stimulus vectors **u** and **v** each have a corresponding boundary that divides the weight space into states of high and low response (1 and 0). These two boundaries define four regions that can each be labeled according to the

responses to the two vectors; the bold labels (**00**, **01**, **10**, **11**) thus indicate the responses of any weight state in that region to the vectors **u** and **v**. A threshold value $\theta = 1$ is assumed. A different value of θ shifts the boundaries (without affecting their orientations).

Equivalent units

Any two units having weights in the same region of Figure 1.15 will give identical responses to the patterns **u** and **v**. Thus, with respect to a given (finite) stimulus set, **S**, there are regions of weight space over which the response properties of all units defined by weights in that region are equivalent in that they are indistinguishable by their response properties over **S**. Note, however, that there exist patterns that can distinguish between any two nonidentical[3] units. In general, for a given threshold value and a given set of stimulus patterns, the weight space is "shattered" into several regions (see Figures 1.15 and 1.16).

If the threshold value is *not* fixed (i.e., we are free to choose the weights and the threshold), then it is possible for units with different parameters to be exactly equivalent computationally. This type of equivalence holds for any two units with corresponding parameters that have a constant ratio; that is, if the parameters for unit A are the vector $\mathbf{w}^A$ and θ^A, and the parameters for unit B are $\mathbf{w}^B$ and θ^B, where $\mathbf{w}^B = k\mathbf{w}^A$ and $\theta^B = k\theta^A$, for some positive constant k, the units will be functionally identical (meaning that they will always compute the same response for a given stimulus).

Boolean Tasks

Boolean tasks are a class of functions from classical logic, where the variables can take on one of only two values, **true** or **false**. There are three fundamental Boolean **operators** (i.e. functions) that can be applied to Boolean variables: **NOT**, **AND**, and **OR**. Consider a Boolean variable P, defined to be the proposition that "John will be out of town next Friday", and another variable Q, defined as "John's office's party will be next Friday".

- The **negation**, or **NOT** operator, denoted by either a tilde before the operand variable ($\sim P$), or a bar above it ($\overline{P}$), would be a proposition that is exactly complementary to P; e.g.., "John will **not** be out of town Friday".
- The **conjunction**, or **AND** operator (denoted in the notation of symbolic logic by the symbol ||), combines two propositions (for example, P and Q, above) into a new proposition that is true *only* when the two propositions are both true; e.g., if "John is out of town Friday **and** the office party is Friday".

---

[3]Two linear threshold units, A and B, have identical responses for all stimuli if and only if the thresholds and weight vectors of the units satisfy $\theta^A \mathbf{w}^B = \theta^B \mathbf{w}^A$.

- The **disjunction**, or **OR** operator ( ), combines two propositions (for example, *P* and *Q*, above) into a new proposition that is false *only* when the two propositions are both false; e.g., if "John is out of town Friday **or** the office party is Friday".

The table below specifies the precise meaning of these operators.

Table 1.1. Logical negation, conjunction (| |), and disjunction ( ) operators

P	*Q*	$\overline{P}$ (**NOT** *P*)	$\overline{Q}$ (**NOT** *Q*)	$P \square Q$ (*P* **AND** *Q*)	$P \quad Q$ (*P* **OR** *Q*)
false	**false**	**true**	**true**	**false**	**false**
false	**true**	**true**	**false**	**false**	**true**
true	**false**	**false**	**true**	**false**	**true**
true	**true**	**false**	**false**	**true**	**true**

Neural networks can be used to process Boolean tasks by restricting the values of the input variables and unit responses to two numerical values, assigned to correspond to **true** and **false** respectively. The conventional numerical assignment of 0=false, 1= true, will be used in the following sections.

A **Boolean Task** is simply a particular assignment of true/false values to each possible input, where an input can be specified by any number of Boolean variables. Given *N* input variables, there are *P* (=2^N) input patterns, each of which can be assigned 1 or 0 independently in the definition of a task; therefore there are $2^P = 2^{(2^N)}$ definable tasks.

Boolean tasks with two arguments (2D)

There are $P = 2^2 = 4$ patterns, and hence 16 ($2^{(2^2)}$) Boolean tasks that can be defined over two variables; six of these are shown in the table above; note that the values listed in the "P" and "Q" columns can be interpreted as (almost trivial) functions. The complete set is shown below.

Table 1.2. The complete set of Boolean tasks on 2 variables.

x	*y*	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Many of these Boolean functions of two variables have time-honored names; in particular, the **OR** and **AND** functions (described above). For every Boolean function *f*, there exists a *complement* function ~*f*, which is true for arguments when *f* if false, and false when *f* is true. Thus, the complement of an LS function is also LS, and can be separated with the same boundary (requiring a sign change to the weight and threshold values). The **NOR** (for

Not OR) and **NAND** (Not AND) are the complements of the **OR** and **AND** respectively. The **XOR** (for "exclusive or") function is defined to be **1** (true) if *x* or *y* is true, but not both. The equivalence function ("**=**") is true only when *x* =*y*. One of the functions (f_3) corresponds exactly to the first argument (this is labeled "*x*" in the figure), and that f_5 replicates the second argument ("*y*"). If the two variables are inputs to a unit, the four input patterns ((0,0), (0,1), (1,0), and (1,1)) can be plotted as points in an input space that lie at the vertices of a square (as in Figure 1.16).

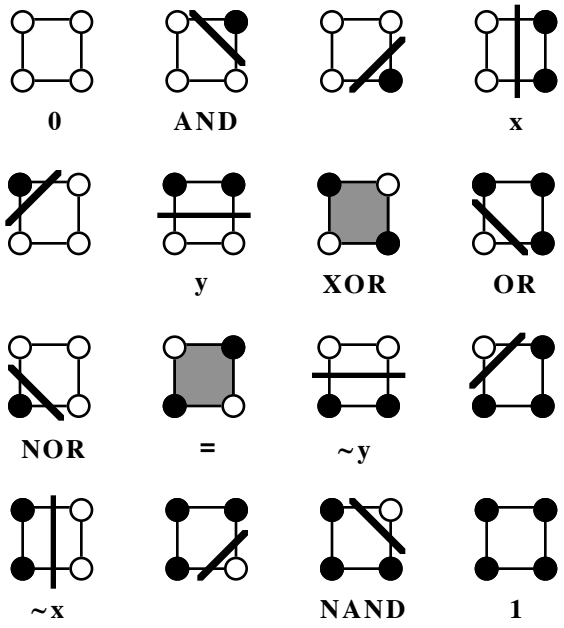


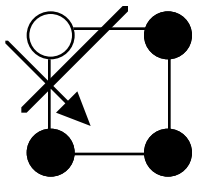
Figure 1.16 The sixteen Boolean functions of two variables viewed in stimulus space. Each square depicts one of the two-variable functions. The vertices (small circles) each represent one of the four sets of arguments ((0,0), (0,1), (1,0), and (1,1)) and the shading of the vertex indicates the function value for that argument (black=1, and white=0). These are the functions f_0 to f_{15} from Table 1.4 arranged from the upper left in rows; i.e. the first row is f_0 - f_3, the second is f_4 - f_7, etc. Lines separating the two classes (0 and 1) are shown for the 14 functions that are linearly separable; the 2 functions that are not LS are shaded gray.

Note that 14 of the 16 tasks are LS, and only 2 are non-LS, **XOR** (f_6) and its complement, the equivalence function (f_9). Since it is the simplest Boolean function that is not LS, the **XOR** task has been analyzed ad nauseum (as we will see).

Consider how the four Boolean patterns partition the weight space. Setting the threshold θ equal to 1, and using +1 and -1 for the stimulus values, the picture in Figure 1.17 is obtained (-1 is used rather than 0, only for aesthetic purposes, since the resulting figure is symmetric with respect to the axes). The reader should note that since each boundary represents the response to a single pattern, labels in adjacent regions differ in just one bit.

Example 1.4 Find a set of weights and a threshold corresponding to the function f_{11}.

As in Example 1.3, we first need to determine a weight vector. First plot the function in order to visually determine the direction of the weight vector, then calculate the resulting weighted sums to determine a threshold value. The function f_{11} has a value **0** for the stimulus pattern (0,1) and a value **1** for the other 3 Boolean stimuli, which results in the following figure:



The weight vector **w** = (1, -1) is seen to lie perpendicular to a separating boundary, giving the weighted sums in the table, so we see that a threshold value between -1 and 0 will give us the desired response.

Stimulus		Weighted Sum	Response
0	0	0	1
0	1	−1	0
1	0	1	1
1	1	0	1

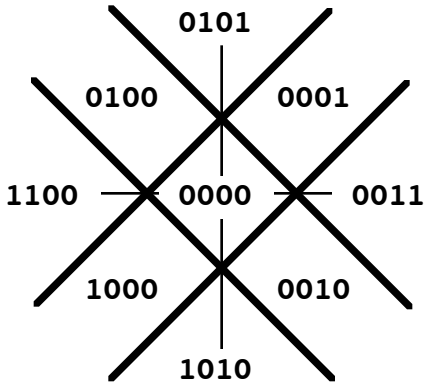


Figure 1.17. Weight Space Partitioning by an LTU with θ=1 for Two Boolean Inputs. The 4-bit label in a each region lists the responses to the 4 Boolean input patterns in the sequence: (-1,-1), (-1,+1), (+1,-1), (+1,+1).

Note that only 9 of the 14 LS functions appear in Figure 1.17. Note that the 5 functions that do not appear all have more **1** responses than **0** responses. Thus, it can be concluded from this diagram that those 5 functions cannot be computed if θ =1. The functions can be computed by setting θ = 1 (see Exercise 5 at the end of the chapter).

Note that only 9 of the 14 LS functions appear in Figure 1.17. Note that the 5 functions that do not appear all have more **1** responses than **0** responses. Thus, it can be concluded from this diagram that those 5 functions cannot be computed if $\theta = 1$. The functions can be computed by setting $\theta = -1$ (see Exercise 5 at the end of the chapter).

Boolean tasks with more than two arguments (>2D)

A linear threshold unit with three inputs and a given set of parameters (weights and bias) divides the 3D stimulus space in half. For the weight vector in Figure 1.4, any of the orthogonal planes could be the separating plane, depending on the threshold value. The side of the plane toward which the vector is pointing is the region of stimulus space that will generate a "high" response ($r=b$). The behavior of linear threshold units in higher dimensions is analogous.

For three stimulus variables, there are $P=2^3=8$ patterns, and hence 256 ($2^{(2^3)}$) Boolean tasks that can be defined. Unlike the 2D case, where nearly all (14 out of 16) of the Boolean functions are linearly separable, less than half of the 256 3D functions are LS. As a function of N, the number of LS Boolean functions $L(N)$ grows much more slowly than $P(N)$, thus linearly separable functions become rarer with increasing dimensionality. Thus the fraction of possible functions that are computable by a single unit becomes vanishingly small (see Table 1.3). So, except for the smallest values of N, the linear separability restriction is too severe to let stand-alone LTUs be used as general classifiers.

Table1.3. Number of Boolean functions and linearly separable Boolean Functions for N variables

N	$P(N)$	$L(N)$	$P(N)/L(N)$
1	4	4	1.00000
2	16	14	.875000
3	256	104	.406250
4	65536	1882	.028717
5	4294967296	94572	.000022
6	1.84×10^{19}	1.5×10^{7}	$< 10^{-12}$

The single pattern task

Consider the "single pattern" task, defined as a Boolean task on N inputs, for which the response to one particular pattern, **P**, (of the 2^N total patterns) is 1, and the response to all other patterns is 0. No matter which pattern is selected to be **P**, this task is linearly separable. For N=2, or N=3, the linear separability of the single pattern task is obvious by visual inspection, since the patterns correspond to vertices of a square (N=2) or a cube (N=3), any vertex of which can be separated from the others by a plane. While the visual argument fails for N>3 (for most of us, anyway), linear separability in this case can be demonstrated (by construction of a solution) as follows:

Single Pattern Theorem: For any Boolean pattern **P** in N dimensions, there exists a set of parameters (weights **w**, and a threshold θ) for a linear threshold unit, such that the unit will respond only to **P**, and to no other Boolean pattern.

Proof (by construction): Let the weight vector **w** consist of components equal to +1 and -1. Specifically, $w_i = 2P_i - 1$ (i.e. if $P_i = 0$, $w_i = -1$, and if $P_i = 1$, $w_i = +1$). In this case, the weighted sum for **P** is equal to K, where K is defined as the number of 1s in **P**. Any other pattern must have a lower weighted sum, since a change for any pattern component to **P** reduces the weighted sum by 1. Hence a threshold value between K-1 and K will do the trick.

Example 1.5. Find a set of weights and a threshold that respond only to the Boolean pattern **P=(1 0 1 1)**, and to no other. According to the construction in the proof, **w** is chosen to be **(1 -1 1 1)**, which gives a weighted sum of 1+0+1+1 = 3 to the given pattern. The weighted sum for every other pattern is reduced by an amount 1 for each bit that is different from **P**. The weighted sums for all 16 patterns are shown below:

sum = 3	sum = 2	sum =1	sum = 0	sum = -1
1 0 1 1	**0 0 1 1**	**1 0 0 0**	**0 1 0 1**	**0 1 0 0**
	1 1 1 1	**1 1 1 0**	**0 1 1 0**	
	1 0 0 1	**0 0 1 0**	**0 0 0 0**	
	1 0 1 0	**1 1 0 1**	**1 1 0 0**	
		0 0 0 1		
		0 1 1 1		

Since all other patterns give a sum of 2 or less, θ can be chosen to be 2.5.

The 2-D Boolean functions have analogues in higher dimensions. The n-bit versions of **0**, **1**, **OR**, **AND**, **NOR**, and **NAND** are easily defined (for example **OR***n* is true if and only if *any* of the n bits are 1):

s	0	OR*n*	AND*n*	NOR*n*	NAND *n*	1
000...00	0	0	0	1	1	1
000...01	0	1	0	0	1	1
⋮	0	1	0	0	1	1
111...10	0	1	0	0	1	1
111...11	0	1	1	0	0	1

The *n*-bit excusive or (**XOR***n*) and the *n*-bit parity task (**PAR***n*) defined below, are both *n*-dimensional extensions of the **XOR** task; or put another way, both tasks reduce to **XOR** in the case where *n*=2. The case *n*=3 is illustrated in the table.

s	XOR3	PAR3
000	0	0
001	1	1
010	1	1
011	1	0
100	1	1
101	1	0
110	1	0
111	0	1

1.3. Semi-linear Units

Mathematical definition

A semilinear unit has a transfer function that is a scalar function of the weighted sum of the inputs (see Figure 1.18). Two examples of semilinear units have already been presented in this chapter, the linear unit, and the linear threshold unit. A new parameter, called the unit bias, is introduced here (denoted *b*).

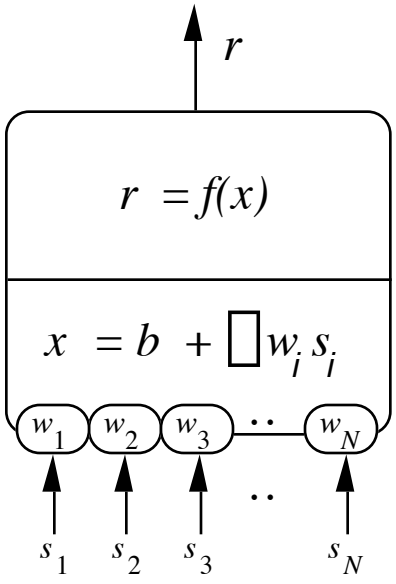


Figure 1.18. A semi-linear unit. The weighted sum (*x*) is computed and passed through the nonlinear function *f(x)*. The unit typically has *N* weight parameters and a bias parameter (*b*).

A linear unit is a special case of the semi-linear unit: set b to zero, and choose $f(x)$ to be the identity function, $f(x)=x$. The linear threshold unit is another special case, even though there is (apparently) no explicit parameter corresponding to the threshold. By defining $f(x \geq 0)$ to be 1 and $f(x < 0)$ to be 0, and setting $b=$ [illegible], the unit corresponds exactly to the LTU (the demonstration of this is left to the reader).

The PASCAL procedure below computes the output for a semilinear unit:

```
procedure semixfer(stim:Array[1..maxin] of Real,  {Stimulus pattern}
                   wts: Array[1..maxin] of Real,   {Weight vector}
                   bias: Real ;                    {Bias}
                   fanin:Integer)      {The number of inputs}

var    ii : Integer ;
       x : Real :
begin
    x := bias ;
    for ii := 1 to fanin do begin
       x := x + wts[ii]*stim[ii]
       end ;
    semixfer := bfunc(x) {bfunc is defined elsewhere to be whatever function is
desired (such as the logistic)}
end;
```

Since the response of a semi-linear unit is a function of the linear sum, it can be inferred that any contour where the linear sum is constant is also a contour of constant response. Hence, as in the case of the linear unit, the contours of constant response are hyperplanes (lines in 2D, planes in 3D, etc) oriented orthogonally to the weight vector (and hence parallel to each other). However, the spacing between the contours is different than for the linear unit, as will be seen for the following function types.

Sigmoidal functions

As will be seen in Part II, the choice of the unit transfer function must be done carefully. In the backpropagation procedure, all unit transfer functions must be *differentiable*[4]. Other important considerations in choosing the unit transfer function are whether it is bounded (never exceeding certain prespecified upper and lower limits) and whether it is monotonic (always increasing or decreasing in their argument). *Bounded* functions are functions which to not get arbitrarily large (either in the positive or negative direction); that is they satisfy the condition that there exist bounds, L_{lower} and L_{upper}, such that $L_{lower} < f(x) < L_{upper}$ for all x. *Monotonic* functions are always increasing (monotone increasing) or always decreasing (monotone decreasing); that is, the derivative $f'(x)$ is either always positive or always negative. While boundedness and monotonicity are not essential to backprop, in most

[4]The function only needs to have a first derivative; the existence of higher order derivatives is irrelevant. Mathematicians refer to such functions as C_1 (where C_n refers to a function that can be differentiated n times).

cases they are desirable. A class of functions called "sigmoid" (Greek for "s-shaped") satisfies the differentiability, boundedness, and monotonicity criteria. Examples are the arctangent and hyperbolic tangent functions; the most common function used in neural network processing (especially in backprop networks) is the *logistic* function $L(x)$, defined in Equation 1-20.

$$L(x) \equiv \frac{1}{1 + e^{-x}} \qquad [1\text{-}20]$$

A sigmoid transfer function is sometimes chosen as a differentiable surrogate for a threshold function, particularly when the output unit(s) are computing a binary value (such as a classification task). Consider the lines of equal response for $L(x)$ (Figure 1.18). Note that the lines corresponding to minimum and maximum values of the response (eg, $r = 0$ and $r = 1$ for the logistic) lie at infinity. The intermediate response (r =0.5 for the logistic) line is sometimes treated as if it were the 0-1 threshold boundary from the logistic; that is, for a binary task, $r > 0.5$ would be interpreted as 1, and $r < 0.5$ as 0.

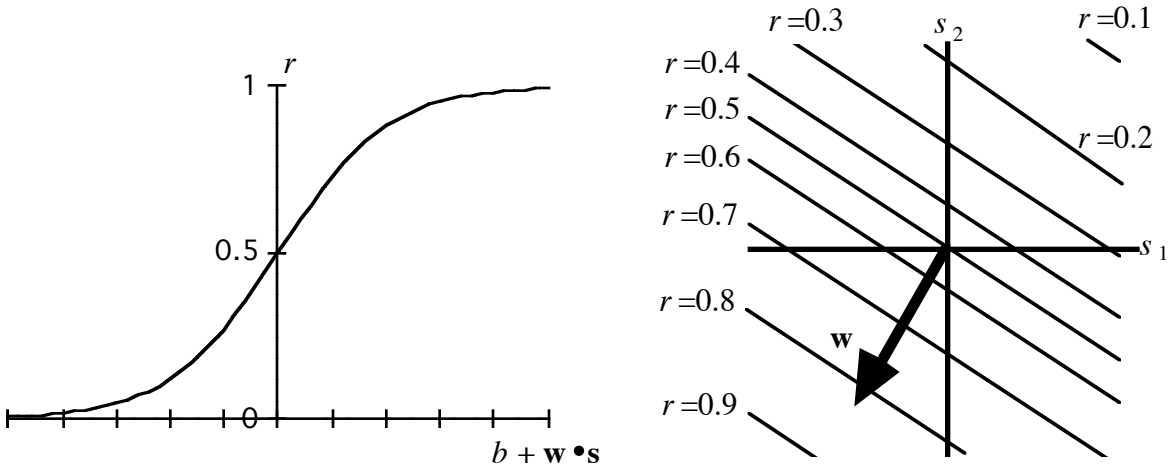


Figure 1.19. Isoresponse contours for a sigmoidal unit. Left: The function $L(x)$. **Right:** The contours all lie in a common direction, orthogonal to the weight vector, but the spacing at uniform intervals of r is not at uniform intervals in weight space.

Approximating the sigmoid by a ramp function of the form in Equation 1-21, the sigmoid can be conceptualized as having a *linear domain* in which $f(x)$ varies significantly with changes in x (the shaded region in Figure 1.20), whereas $f(x)$ is not very sensitive to changes in x outside the linear domain.[5] The breadth and location of the linear

[5] There is not a single definition for the "linear range". One common definition is to bound the linear range at the values of x where the sigmoid has the greatest *curvature*. See Exercise 10 at the end of this chapter.

range is often crucial; as demonstrated in the example below, the unit can exhibit response properties that are either threshold-like or graded, depending on the choice of the function parameters.

$$f(x) = \begin{cases} 0 & x < \theta_1 \\ \dfrac{x - \theta_1}{\theta_2 - \theta_1} & \theta_1 \le x < \theta_2 \\ 1 & x \ge \theta_2 \end{cases} \qquad [1\text{-}21]$$

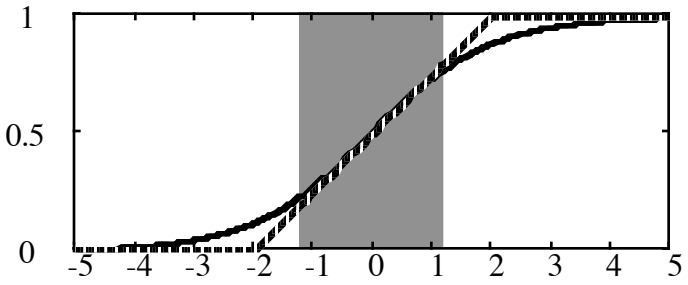


Figure 1.20. The ramp function as an approximation to a sigmoid. A ramp function ($f(x)$ defined by Eq. 1-21) is plotted with the sigmoid $L(x)$, with threshold values chosen (θ_1=-2 and θ_2=2) to match the value of the derivative at x=0. In the vicinity of x=0 (shaded region), the function is well approximated by the line y=(x+2)/4.

Consider a sigmoidal unit with a single input; that is, the unit's response, y, as a function of the input signal, x, the single weight parameter w, and the bias parameter b, is the sigmoid $y = L(wx + b)$. Note that, for smaller values of w, the linear range of the unit extends over a broader extent in the x domain, and that as w increases toward larger values, the linear range shrinks, approaching a threshold function (see Figure 1.21).

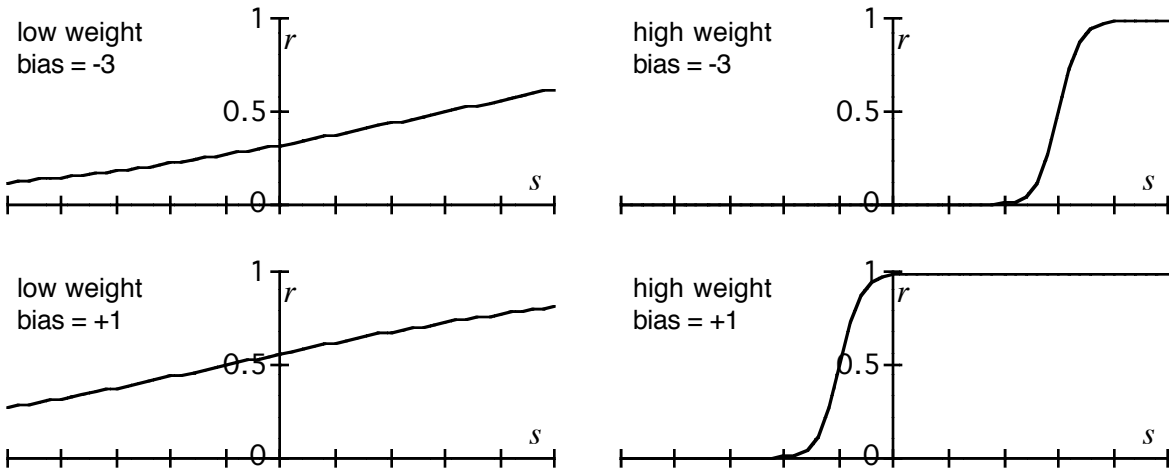


Figure 1.21. Effect of the weight and bias parameters on the linear range. As w increases, the sigmoid approaches a threshold function, decreasing the width of the linear range. Varying the bias parameter (b) shifts the linear range.

Nonmonotonic functions

It is sometimes useful to consider bounded functions that are *unimodal*[6] (as opposed to monotonic) for use in semilinear units; i.e. functions $f(x)$ that have a single maximum (x_{max}). That is, the derivative $f'(x)$ is positive for x

Example 1.6. For a sigmoidal unit with a single input, find values for the weight and bias parameters that give a graded, roughly linear response from 0.2 to 0.8 for inputs from 0 to 10.

Even though the sigmoid function is nonlinear, this problem can be made linear by finding the weighted sum values that correspond to responses of 0.2 and 0.8. Thus, if $x=ws+b$, then $r=f(x)$, and $x=f^{-1}(r)$, then we can get a value for x by inverting the logistic function:

$$r = L(x) = \frac{1}{1+e^{-x}} \quad \Rightarrow \quad x = L^{-1}(r) = -\ln\left(\frac{1-r}{r}\right) = \ln\left(\frac{r}{1-r}\right),$$

where the natural logarithm $\ln(x)$ is the inverse of the function e^x. This gives two equations: x should be $\ln(.2/.8) = -1.386$, when s is 0, and x should be $\ln(.8/.2) = 1.386$ when s is 10:

$$b = -1.386$$
$$10w + b = 1.386$$

Thus, we obtain $w= 0.2772$ and $b= -1.386$.

$<x_{\max}$ and $f'(x)$ is negative for $x > x_{\max}$. (hence $f(x) < f(x_{max})$ for all $x \neq x_{max}$). The Gaussian function is the classic example of a unimodal function. Figure 1.22 shows the constant response contours for a semi-linear unit using a Gaussian function. Another interesting possibility is a periodic function, such as a sine or cosine (Figure 1.23).

---

[6] A unimodal function $u(x)$ has a single local maximum (a single "peak"). More formally, this can be stated as follows: there is only one value of x for which $u(x)$ is greater than all values of $u(x\pm\epsilon)$ no matter how small ϵ is, as long as $\epsilon>0$. In a *bimodal* function there are two such points.

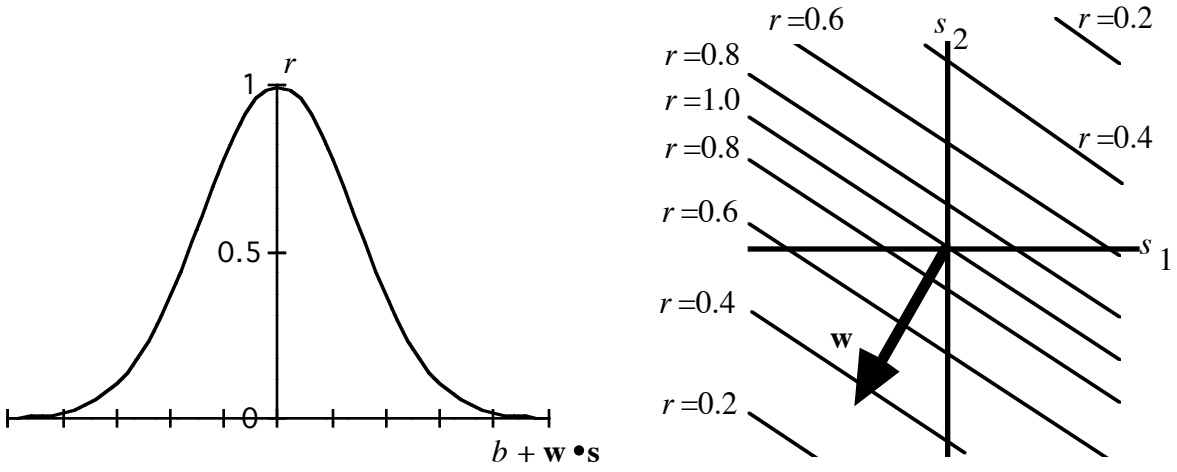


Figure 1.22. Response contours of nonmonotonic semilinear units. As in the case of units with monotonic functions, the contours of constant response are aligned orthogonal to the weight vector and thus parallel to one another.

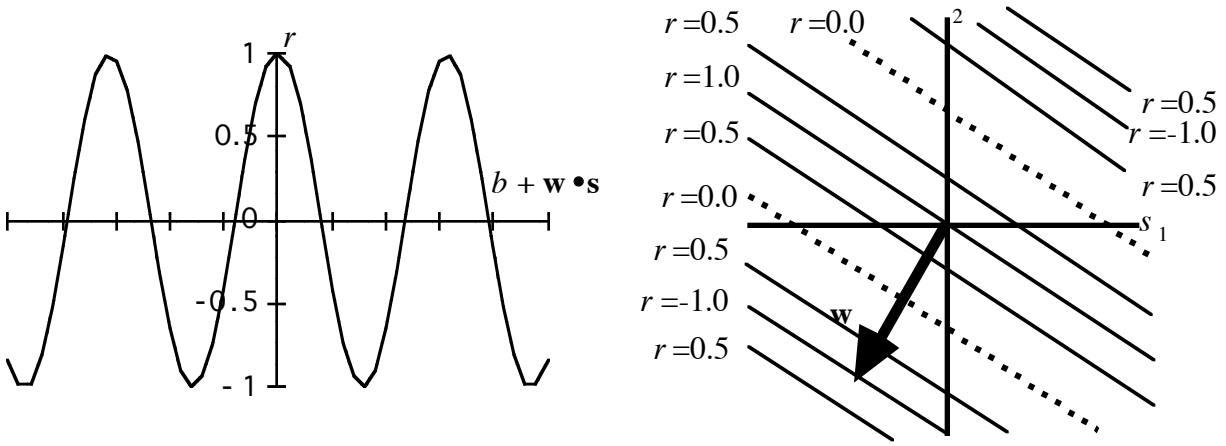


Figure 1.23. Response contour of a periodic semilinear unit. Again, the contours are parallel. In this case, the cosine, they form "banded regions" of high response that are not connected. The dashed contours indicate $r = 0$.

1.4. Other (Sigma-Pi, Radial, etc)

While semi-linear functions are typically chosen for feed-forward neural network architectures, the restriction that isoresponse contours must be linear is sometimes unsuitable, hence other classes of functions are sometimes used. The most common exceptions to the semi-linear assumption are described in this section.

Radial

Among these other classes, the most prevalent are functions that respond maximally to an optimal stimulus vector **s***, with a response that decreases as the stimulus vector is more distant from **s***. Such a unit can be defined by first computing the distance between the stimulus **s** and the optimal stimulus, **s***. Typically, a Euclidean distance function is used: $d = ((\mathbf{s}^*-\mathbf{s})\bullet(\mathbf{s}^*-\mathbf{s}))^{1/2}$. The equal response contours in this case are "hyperspheres" (circles in 2D, spheres in 3D, etc). The distance can then be passed as an argument to another function; thus the response would be $r=f(d)$, where f is typically a unimodal function which has its maximum value at $d=0$. The parameters of such a unit are the components of **s*** (there are no "weights" per se). The class of radial units can be described by analogy with the semi-linear units; the distance calculation corresponds to the weighted sum, to which another function is (typically) applied. In both cases, the shape of the equal response contours is determined by the first calculation (weighted sum vs. distance). A 2D radial unit is shown in Figure 1.24 with its isoresponse contours in stimulus space.

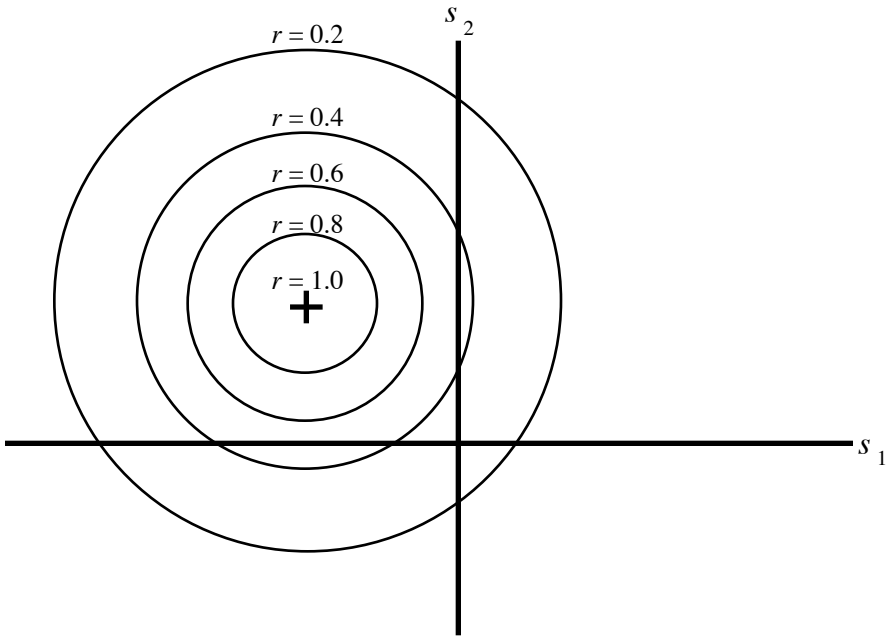


Figure 1.24. Contours of constant response for a radial unit. The parameters of the unit are the coordinates of the central point **s*** (marked by +).

Sigma-Pi

The linear sum can be extended to a second order polynomial:

$$r = \sum_i \sum_j w_{ij} s_i s_j \, .$$

Or to a third order polynomial:

$$r = \sum_i \sum_j \sum_k w_{ijk} s_i s_j s_k .$$

Or to higher orders, using the general formula for an m-th order polynomial:

$$r = \sum_{i_1} \cdots \sum_{i_m} \left(w_{i_1 \cdots i_m} \prod_{k=1}^{m} s_{i_k} \right),$$

The product symbol $\prod$ denotes a product[7], in precise analogy to its more well-known cousin, the summation symbol, $\sum$. Units with this class of function are called $\Sigma\Pi$ (Sigma-Pi), named for the symbols $\sum$ and $\prod$ in the formula. Note that there are many more weight parameters here, N^m for an m-th order unit in N dimensions; there is a separate weight value for every combination of m weights. The notion that a neuron is able to maintain independent parameters for every pair (or triple, or m-tuple...) of stimulus components seems biologically implausible. On the other hand, some investigators interested in exploring units with more complex discrimination boundaries have explored $\Sigma\Pi$ units. The increased computational power of $\Sigma\Pi$ units comes at a cost however, since the large number of free parameters can hurt the generalization performance of the system (this issue will be discussed later).

Example 1.7. Find parameters for a 2nd order $\Sigma\Pi$ unit to compute the **XOR** function.

The function computed by a 2nd order $\Sigma\Pi$ unit with two inputs, s_1 and s_2, is $r = w_{11}s_1^2 + w_{12}s_1s_2 + w_{22}s_2^2$. Note that the w_{21} term has been omitted, since it is redundant with the term w_{12}. The unit can thus be treated like a linear unit with three inputs.

Stimuli		Products			Target
s_1	s_2	s_1^2	s_1s_2	s_2^2	***XOR***
0	0	0	0	0	0
0	1	0	0	1	1
1	0	1	0	0	1
1	1	1	1	1	0

Thus, the weights $w_{11} = w_{22} = 1$ and $w_{21} = -2$ compute **XOR**.

[7] That is, $\prod_{i=1}^{N} x_i \equiv x_1 x_2 \cdots x_N$.

Product Units

Durbin & Rumelhart (1989) define units that compute products of inputs, but combine them differently than in the above case of □□ units; these could be called simply *product units* (or □ units). The modifiable parameters here are *exponents* applied to the input values, rather than multiplicative weights:

$$r = \prod_{i=1}^{N} s_i^{p_i} \qquad [1\text{-}22]$$

Note that this function is not as far from the linear sum as it might seem, since taking the logarithm of both sides yields:

$$\log(r) = \sum_{i=1}^{N} p_i \log(s_i) \qquad [1\text{-}23]$$

A potential problem arises if a stimulus component s_i is negative, and the corresponding parameter p_i is not an integer. Raising negative numbers to a power that is not an integer requires complex numbers.[8] Of course, one could restrict the values of the stimulus components to be greater than zero. Durbin and Rumelhart (1989) argue that "... the non-linear characteristics of product units, which we want to use computationally, are centered on the origin." (p.135). The response r is computed as a complex number (Eq. 1-24); only the real part is used for computation.[9]

$$r = e^{\sum p_j \log|s_j|}\left(\underbrace{\cos\left(\pi \sum_{k|s_k<0} p_k\right)}_{\text{real part}} + i\underbrace{\sin\left(\pi \sum_{k|s_k<0} p_k\right)}_{\text{imaginary part}}\right) \qquad [1\text{-}24]$$

Some notable properties of the product unit response function:

---

[8] The so-called "imaginary number" i is defined as $(-1)^{1/2}$, which is the same as $\sqrt{-1}$. Standard mathematical functions, can be defined in terms of the system of complex numbers, $z = x + iy$, where x and y are real numbers. Each complex number z thus has a real part, x, and an imaginary part, iy. In the system of complex numbers, functions like the square root and the logarithm, which are undefined for certain domains of the real range, are "complete"; that is $f(z)$ can be computed for and complex number z. A key identity in this system is the exponential function of an complex number $e^{x+iy} = e^x (\cos y + i \sin y)$.

[9] Durbin & Rumelhart (1989) mention two possible approaches for dealing with the imaginary portion of the response function; namely, it can either be fit to zero (i.e. minimized as part of the cost function) or ignored. They opt for the latter in their simulations.

- Unlike sigma-pi units (described earlier in this section), which require many more parameters than a semi-linear unit, the number of parameters is the same as it is for a semi-linear unit, yet is has the potential to capture high order functions of the inputs (like ratios).
- If all stimulus components s_k are positive, the complex number in the large brackets of Eq. 1-24 reduces to 1.
- The contours of constant response are hyperbola-like curves (Figure 1.25).

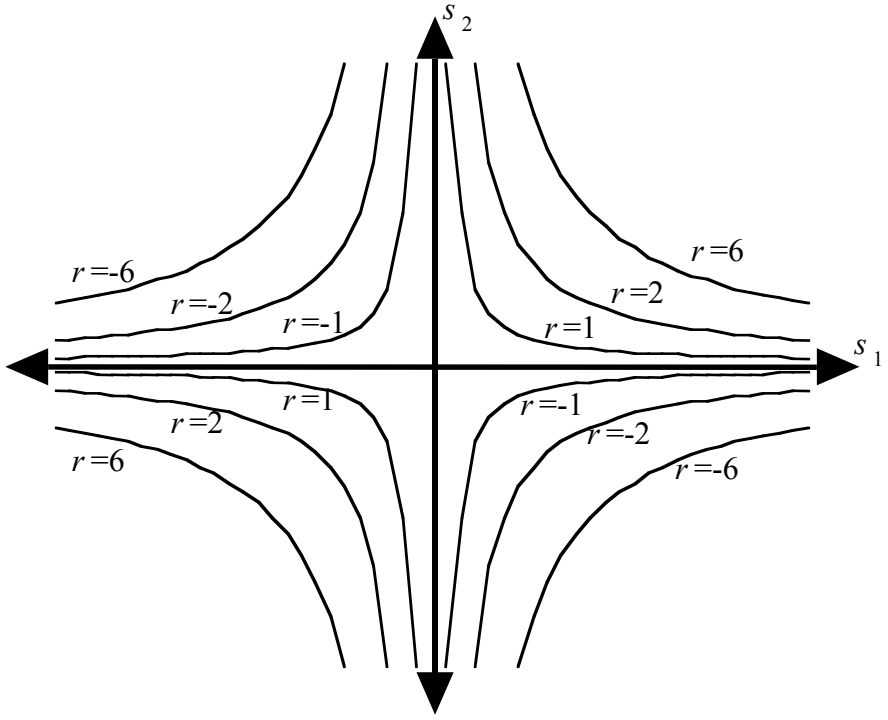


Figure 1.25. Isoresponse contours for a product unit. The parameters for this unit are $p_1=p_2=1$.

Stochastic

Many neural network models require some degree of randomness. Units with responses that are not entirely determined by the input stimulus and the unit parameters are called *stochastic*. It is generally not desirable to have a response that is 100% random; such a response would be independent of the stimulus. A stochastic unit generates a response that is neither entirely deterministic nor entirely random. Generally, the response is drawn from a random distribution, whose parameters (for example the mean of the distribution) are influenced by the stimulus. For example, a given stimulus vector will not yield the same response on every presentation to a stochastic unit with a given set of parameters, yet certain stimuli will have a *tendency* to generate a stronger response than others. Of course, there are several techniques for introducing random influences to the unit transfer function.

Two of these are illustrated in Figure 1.26.

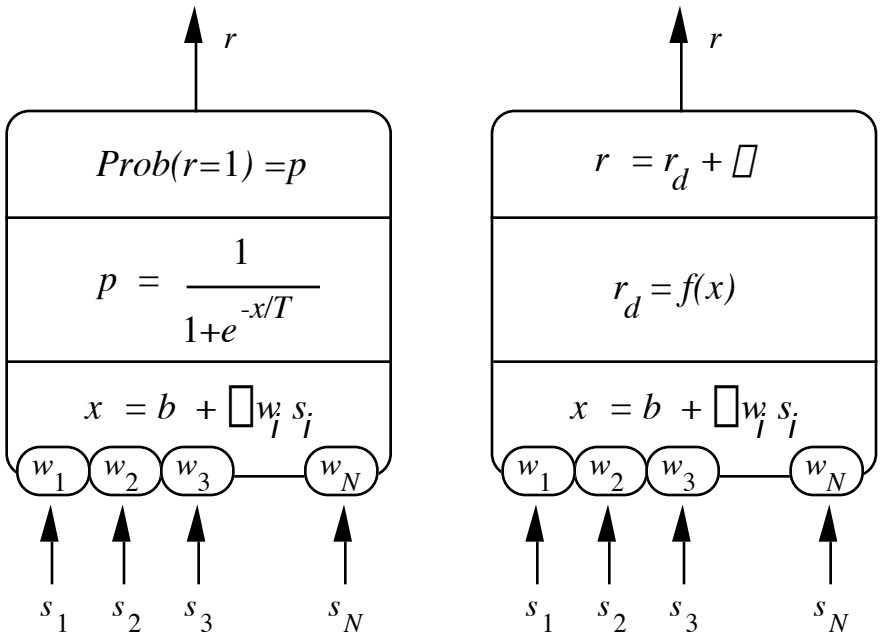


Figure 1.26. Two kinds of stochastic units. Left: A Boltzmann Unit (Ackley, Hinton, and Sejnowski, 1985) always responds with a value of 0 or 1, but the expected value for a given input is equal to the logistic of the weighted sum, scaled by a "temperature parameter" *T*. **Right:** Random noise added to the output of a semilinear unit. Let ε denote a random value (the distribution of the random number is not discussed here; see Appendix A for an introduction to random value generation). Thus, $r = r_d + \varepsilon$, where r_d is the deterministic value.

1.5. Similarity and Discriminability

A primary function of single units, both in isolation and in the context of a larger network, is to discriminate between patterns that may have similar vector representations (**s**) in some cases, and to give identical responses to vectors that are very different in other cases. The *similarity* between vectors has many possible mathematical definitions (more on these in Section 5.2); for now, let us define similarity as the cosine of the *angle* between the vectors, which is equal to the dot product for unit vectors (i.e., vectors of length 1 unit).[10] Consider two input

[10] The response of a single unit can be viewed as a *measure of similarity between the stimulus vector* (**s**) *and the weight vector* (**w**). This is clearly true for a linear unit (it computes the dot product similarity measure directly), but in a more general sense, any function defined for a unit is a kind of similarity measure between **s** and **w**.

vectors **a** and **b**, with an angle between them of □. Suppose a unit is confronted with the task of responding to **a** with

Examples are shown in Figure 1.27 of two pairs of vectors, one pair with a small angle between them (i.e., very similar), and the other pair almost orthogonal (i.e., dissimilar). Suppose the vectors are to be discriminated by a linear unit, so that the response to **a** is 0, and the response to **b** is 1. The weight vector that satisfies both of these criteria In the next chapter, we will see that weight states that discriminate between similar vectors are not only more distant from the origin, but also that learning the discrimination is a *slower* process, where learning will be described as a trajectory of a weight state over time. Thus the distance of the attractor from the origin is a rough measure of the discrimination difficulty.

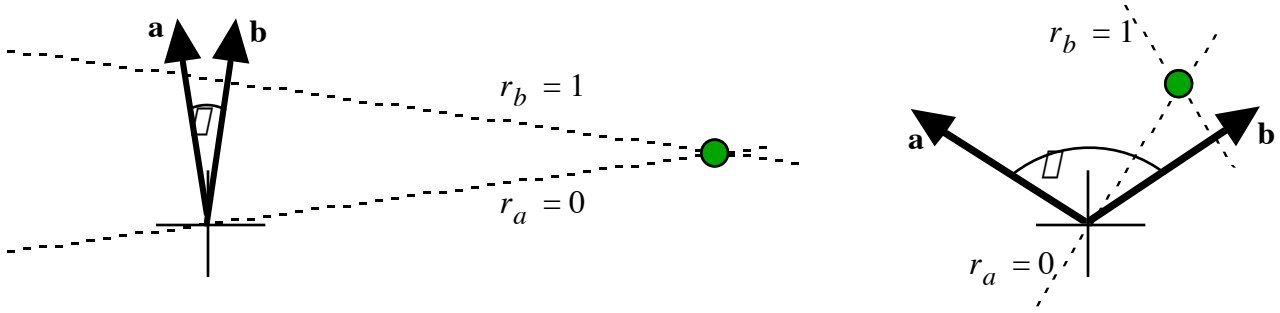


Figure 1.27. Similar vectors are more difficult to discriminate. The similarity (here, as measured by the cos(□)) between two vectors has an effect on their discriminability. Observe that the weight vector that discriminates between them (at the intersection of the dashed isoresponse contours) is pushed to greater distances as □ decreases. Note that the intersection goes to as □ goes to zero.

1.6. Exercises

1. A linear unit with two input components has weights $w_1 = 3$ and $w_2 = -1$. Compute the response of the unit to the following stimuli:
   a. **s** = (0, 0)
   b. **s** = (-2, 6)
   c. **s** = (1, 2)
   d. **s** = (0,-1)
2. For the weights in exercise 1, draw isoresponse contours for r =0, r =1, and r =2.

3. For a linear unit with three inputs, find two different sets of weights that satisfy the stimulus-response constraints given by the following task:

Stimulus	Response
0 4 1	4
-1 2 0	2

4. Determine which of the tasks are LS. Find weights and a threshold for those that are.

a

Stimulus	Response
2 3	0
2 5	0
1 6	1
-1 2	1

b

Stimulus	Response
2 2	0
0 0	0
1 0	1
2 1	1

c

Stimulus	Response
0 4 1	0
-2 1 1	0
2 4 1	1

5. Plot the discrimination boundaries in weight space for the patterns (-1,-1), (-1,+1), (+1,-1), (+1,+1), assuming a threshold value $\theta = -1$. (This is the "flip side" of Figure 1.16).

6. ** Prove that the "neighboring two-pattern task" is linearly separable, where this (Boolean) task is defined as having a "1" response to just two patterns, $\mathbf{s}^{\alpha}$ and $\mathbf{s}^{\beta}$ that differ only in one component, and a "0" response to all other patterns. [For example, $\mathbf{s}^{\alpha}$ = **001101**, and $\mathbf{s}^{\beta}$ = **011101**.]

7. Consider a semi-linear unit defined with a "double threshold", i.e. for $f(x)$ defined in terms of the weighted sum x (with no bias):

$$f(x) \equiv \begin{cases} 1 & \theta_1 < x < \theta_2 \\ 0 & otherwise \end{cases}$$

a. Draw regions of constant response in stimulus space for such a unit, using the weight vector **w**= (1,-1), and threshold values θ_1=2 and θ_2=4.

b. Find a weight vector **w** and threshold values θ_1 and θ_2 that will create an **XOR** unit.

8. Show how a single semi-linear unit using a gaussian nonlinearity can closely approximate **XOR**. Find the appropriate parameters.

9. Show that a single semi-linear unit using a sinusoidal nonlinearity can compute the **PAR***n* task (defined in Section 1.2) for any *n*. [An example of a sinusoid is $r(x) = \sin(x)$.]

10. The *curvature* of a function is defined as its second derivative. By setting the derivative of the curvature equal to zero, determine the two values for *x*, for which the sigmoid $L(x)$ has maximum curvature. A

reasonable definition of the "linear domain" is the range between these values. [Hint: The identity (valid only for $L(x)$), that $L'(x) = L(x)[1 - L(x)]$ may be useful for computation of the derivatives]

11. Find parameters (**s*** coordinates and θ) for a radial threshold unit, defined to give a response $r(d)$=1, if $d<\theta$, and $r(d)$=0 otherwise, that will compute the following tasks. [HINT: by hook or by crook, find a reasonable vector **s***, and then compute the distances. If **s*** is chosen appropriately, finding a threshold is straightforward.]

a

Stimulus	Response
3 3	1
3 1	1
5 2	0
0 2	0

b

Stimulus	Response
2 2 2	1
1 1 1	1
5 6 5	0
-8 0 -8	0

12. The construction in the proof of the single pattern theorem assumes an LTU that responds with **0** or **1**. Repeat the construction for a unit that responds with **-1** or **+1**.

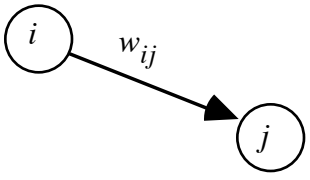
13. Find a weight and bias for a logistic unit with a single input that has an "abrupt" threshold with a linear range that jumps from r =0.05 at s =2.9 to r =0.95 at s =3.1.

2. Neural Networks

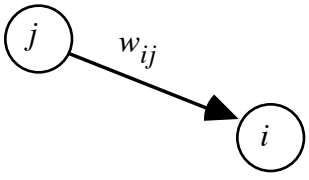
Single units are only of limited interest as stand-alone elements, but when connected into a *network*, the resulting functions have several remarkable properties. Among these is the potential to compute functions that are not linearly separable, even though all the units in the network are linear threshold units. Individual units can, in principle, compute arbitrarily complex functions (for example, by using Sigma-Pi units of arbitrarily high order). An alternative approach to expanding the computational power of units is to combine them into networks; that is the units are "connected", such that the responses of some units serve as stimulus components to other units. The backprop technique is generally applied to this latter type of network, typically composed of relatively simple (typically semi-linear) units.

Notation convention warning

With the exception of the sigma-pi units, the single units in Chapter 1 had just one subscript on the weight values. In this chapter, we begin to describe situations with more than one computational element; thus, *two* subscripts are required for each weight (to identify the presynaptic and postsynaptic units). Two conventions exist in the literature:



Intuitive convention
(for left-right readers)
w_{ij} = weight from i to j



Algebraic convention
(math/science/engineering)
w_{ij} = weight from j to i

chosen for this book

While the choice is arbitrary, one convention must be chosen and used consistently. For this book, we adopt the algebraic convention both for mathematical notation (w_{ij}) and for computer code (`w[i][j]`).

2.1 Network Connectivity

Networks can be classified as either cyclic or acyclic. A cyclic network is one in which any cycles exist, such that it is possible to follow a closed path from the response of a unit to another unit, the response path of that unit to another unit, and so on, ending up at the first unit. An acyclic network then, is a network with no such closed paths (see Figure 2.1).

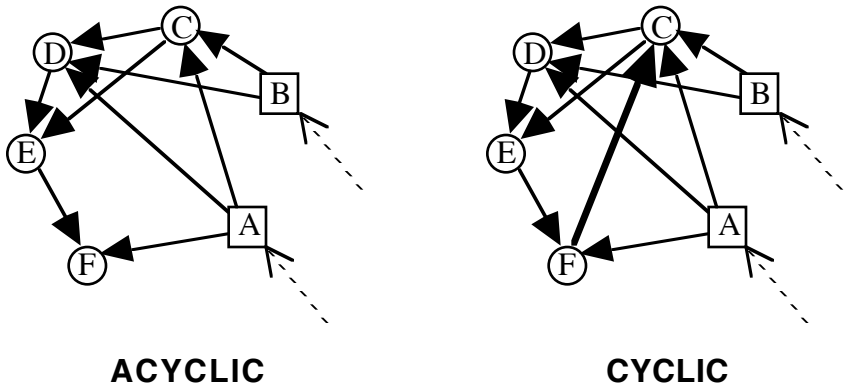


Figure 2.1. An acyclic network (left) and a cyclic network (right). If any closed loops exist in a network, like *C-D-E-F-C* in the network on the right, then the network is termed *cyclic*; a network with no such loops is *acyclic*. Note that the networks are identical, but for the additional link *C-F* (bold) in the cyclic network. The square shaped nodes indicate units that do not receive input from any other units; these *input units* have activities that are determined by the influences outside the network (dashed arrows).

The set of units to which a given unit sends its response is called the *Fanout* set of that unit (or simply, the unit's *Fanout*), and the set of units from which a unit receives its stimulus components is its *Fanin*. For example, the *Fanin* of unit *D* in Figure 2.1 (left) is {*A*,*B*,*C*}, while the *Fanout* of *D* is {*E*}. The connectivity of a network of *N* units can be represented by an *N*-by-*N connectivity matrix* **C**, where the *j*-th element in row *i*, C_{ij}, is 1 if there is a link from unit *j* to unit *i*, and C_{ij} is 0 otherwise. Here the units in the *Fanout* of a given unit, *k*, are those units *i* for which C_{ik} =1, and the units belonging to the *Fanin* of *k* are those units *j* for which C_{kj}=1. For example, the matrix for the acyclic network in Figure 2.1 is:

$$\mathbf{C} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

A corresponding weight matrix **W** can be defined, where w_{ij} is the weight of the connection from j to i. Defining both **C** and **W** may seem redundant; after all, the **W** matrix would seem to contain all the information necessary to compute the output of each unit in the network, where w_{ij}=0 if there is no connection. However, there is an important distinction between no connection and a connection of zero weight. Even though the responses of the nodes (and hence the network) are identical, the learning procedures to be discussed are based on *changing the values of the weights*. Thus, a connection that exists but has weight zero (and may potentially change) must be distinguishable from a weight that does not exist (and hence cannot change).

2.2 Some Observations about Acyclic Networks

The focus of this book, the backpropagation learning procedure, can be applied only to acyclic (also called *feed-forward*) networks, and so let us focus on this class. The following properties of feed-forward networks merit consideration:

- Every acyclic network must have *input units*, which are units that do not receive input from other units (these are denoted in Figure 2.1 by square shaped nodes).
- Likewise, there must be *output units*, in an acyclic network, which are units that do not stimulate other units.
- In a connectivity matrix, the rows corresponding to an input unit are all zero, and the columns corresponding to an output unit are all zero.
- Recognizing that the form of the connectivity matrix depends on how the units are assigned an "order" (unit 1, 2, 3, 4... for purposes of subscripting), there exists an ordering of the units in an acyclic network, such that all upper diagonal values are zero. Of course this includes the diagonal elements, which must be zero, in an acyclic network's connectivity matrix regardless of the ordering.
- Reversing the direction of every connection in an acyclic network results in another acyclic network, in which the input units and output units have reversed roles.

Acyclic networks must have some units that do not have any stimulus components that come from other units; that is, some unit or units receive no activity from no other units (the square node shapes in Figure 2.1 indicate such units). These units are called *input units*, and are not computational elements; that is they do not respond as a function of their inputs (they have none); they generate activity levels that serve as stimulus components to other units in the network. A neurobiological analogue to input units would be the light sensitive detector cells in the retina, which transduce light stimuli to an electrical signal of the type used by neurons. Units **A** and **B** in Figure

2.1 are input units (hence the square shape of the nodes); note that the corresponding rows have no 1s (indicating no connections from other units).

Example 2.1. Determine the weights for a 2-2 layer of linear units that can compute the following stimulus-response mapping:

Stimulus	Response
3 −2	−1 2
1 3	2 −3

Each response component can be treated independently, so for the first component, only the following single unit task need be considered; thus the weights for the first output unit, w_{11} and w_{12}, must satisfy the equations

$$3w_{11} - 2w_{12} = -1$$
$$w_{11} + 3w_{12} = 2$$

The values of w_{11} and w_{12}, are thus -1 and +1 respectively (by elementary algebra). Similarly, w_{21} and w_{22}, must satisfy the equations (the solution is left to the student):

$$3w_{21} - 2w_{22} = 2$$
$$w_{21} + 3w_{22} = -3$$

2.3 Multiple Output Units (single layer)

If a single unit is considered the simplest (albeit trivial) network, the first step beyond this would be a *layer* of units, where a *layer* is defined as a set of units each receiving a common set of stimulus values (see Figure 2.2). Conceptually, this is a minuscule step beyond a single unit, however it has some notational ramifications that are worth mentioning. First, consider a layer of linear units; let r_i denote the response of the i^{th} unit. Not only do the response values, now require subscripts, but the weights which had a single subscript in the single unit case, require two subscripts in the case of multiple units; the weight on the i^{th} unit for the j^{th} stimulus component is denoted w_{ij}. Thus the value r_i can be written as a sum:

$$r_i = \sum_j w_{ij} s_j \quad \text{or} \quad \mathbf{r} = \mathbf{W}\mathbf{s} \qquad [2\text{-}1]$$

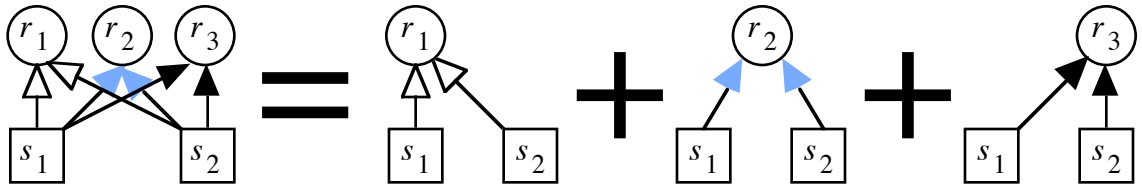


Figure 2.2. A one-layer network. Two units each receive the same input values, but give different responses. Each output response r_i is computed using a set of independent parameters on a common stimulus vector. Squares denote input components, and circles denote computational elements (here, the output units). A single layer of units is simply a collection of independent units; there is no interaction between them and hence no increase in computational power per se.

In vector notation, this equation can be expressed as a matrix multiplication, where **r** is a vector in M dimensions, **s** is a vector in N dimensions, and **W** is an M-by-N matrix: **r**=**Ws**.

Note that a layer of linear threshold units would require a threshold value θ_i for each output unit. For semilinear units, the responses can no longer be computed as a simple matrix multiplication, but the linear sums are simply

$$x_i = b_i + \sum_j w_{ij} s_j ,$$

where b_i is the bias value for unit i. In vector notation, the set of bias values is denoted by the M-vector **b**, giving **x**=**b**+**Ws**. The response values are then computed as $r_i = f(x_i)$, where it is assumed that all the units are using the same function f; otherwise, the different functions could be denoted by a subscript, as in $r_i = f_i(x_i)$. The computer code below computes responses for a layer of semilinear units (having identical functions f).

The PASCAL procedure below computes the output for a layer of semilinear units:

```
function laysemi (stim:Array[1..maxin] of Real,            {Stimulus pattern}
                  wts: Array[1..maxout;1..maxin] of Real,  {Weight matrix}
                  bias: Array[1..maxout] of Real,          {Bias}
                  nin:Integer,                          {The number of inputs}
                  nout:Integer,                         {The number of outputs}
                  ):Array[1..maxout] of Real;

var  ii,jj : Integer ;
     x : Real ;
begin
    for ii := 1 to nout do begin
      x := bias[ii] ;
      for jj := 1 to nin do
```

```
        x := x + wts[ii,jj]*stim[jj]
        laysemi[ii] := bfunc(x)       {bfunc is defined elsewhere to be whatever
       function is desired, such as the logistic}
      end ;                            {ii loop}
end;
```

Example 2.2. Determine whether the following mapping is computable by a 2-3 layer of linear threshold units:

Stimulus	Response
0 0	0 0 1
0 1	1 1 0
1 0	1 1 0
1 1	1 0 0

To decide whether a particular mapping could be computed by a layer of linear threshold units, one need but analyze each output component as an independent classification task, and check for linear separability. If all the output units are computing LS tasks, the entire mapping can be computed (by assigning the appropriate weights and threshold to each output unit); if any of the response functions is not LS, then the entire mapping is not computable by this network. The first component of the outputs (corresponding to r_1) is the **OR** task and is hence LS, as is the third component (r_3); however the second component (r_2), the **XOR** task is not LS, hence the mapping from **s** to **r** cannot be computed by this network.

2.4 Hidden Units

In their 1985 paper on the "Boltzmann Machine" (a neural network procedure), Ackley, Hinton, and Sejnowski introduce the notion of "visible" and "hidden" units in a neural network. *Visible nodes* are those units whose activity levels correspond to observable quantities that are presumably extrinsic to the network (from the environment outside the network). Generally, visible units can be either "input" units (like the detector units in the retina, which transduce light intensity to neuronal activity) or "output" units (like the motor neurons in the spine, which directly stimulate muscle tissue). In the case of feed-forward networks, care should be taken when discussing a network's stimulus values; while these are sometimes referred to as the activity of "input units", we will not use that term here; instead we will reserve the term "unit" to refer to computational nodes in a neural network (circles in Figure 2.3), while the stimulus components to a network will be referred to as "input *nodes*" in the network (squares in Figure 2.3). Visible nodes have interpretable meanings; for example, the input nodes might correspond to the pixels of an x-ray image that is being classified by a neural network as to whether the image contains a tumor

(the presence or absence of which is represented by the output node(s)). *Hidden units* are those units that are not output units; the responses computed by hidden units only activate other units and do not have predefined interpretable meanings.

Consider a network consisting of two layers, with the first layer receives the inputs, and the second layer computes the network response, with only the first layer responses as input; that is, there are no connections directly from the input components to the output units (see Figure 2.3). Let h_j denote the response of the j^{th} hidden unit, let v_{jk} be the weights into hidden unit j from the k^{th} input component; similarly, r_i denotes the response of output unit i, and w_{ij} is the weight matrix from the j^{th} hidden to the i^{th} output unit. Since each of the hidden units and the output units are LTUs, threshold values need to be defined for them.

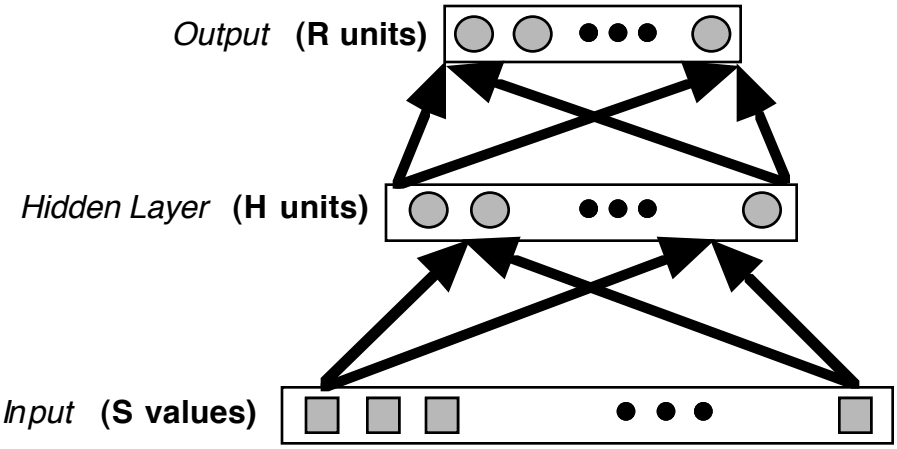


Figure 2.3. A network with a hidden layer. The R output units of this network are stimulated by the H units in the hidden layer which receive stimulation from S input values. Circles denote computational elements and squares denote values provided by the environment. The crossed bold arrows represent full connectivity between successive layers; i.e. all components of one layer contact all components of the next.

A common architecture is to arrange the network in layers; a network with units in well-defined layers, with connections from a given node only to units in the next layer (and not to units beyond that) is called *strictly layered*. Adjacent layers are *fully connected*; that is, all units receive connections from the nodes in the previous layer. The activity of unit i in layer k, denoted $h_i^{(k)}$, is thus computed as a function of the activities in layer k-1:

$$h_i^{(k)} = f\left[b_i^{(k)} + \sum_{j=1}^{H_{k-1}} w_{ij}^{(k)} h_j^{(k-1)} \right] \quad \text{where } i \in \{1 \ldots H_k\},\ h_i^{(0)} = s_i, \text{ and } h_i^{(N)} = r_i \qquad [2\text{-}2]$$

Here, the layer is attached as a superscript (in parentheses) to the weight and activity variables (see Figure 2.4).

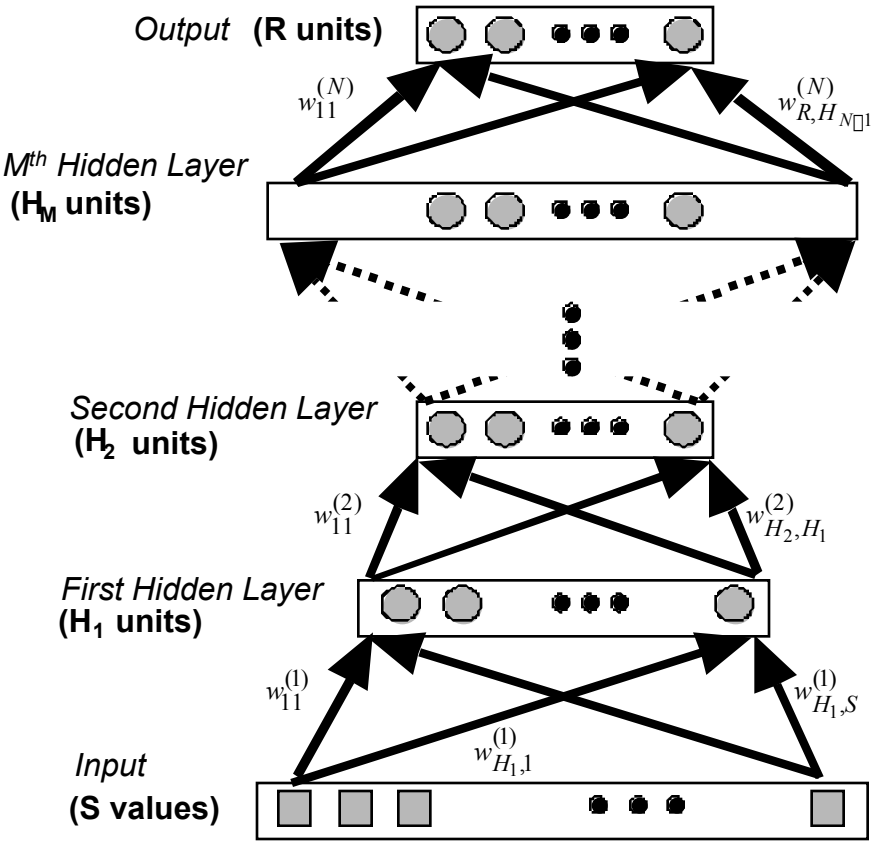


Figure 2.4. A strictly layered network. The general strictly layered architecture can have an arbitrary number of hidden layers. Hidden units in a given layer receive a common set of inputs and send output to a common layer.

The weight matrix of a strictly layered network is sparse (many elements have value zero).

$$\mathbf{w}_{layered} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & \cdots & w_{1,S}^{(1)} & 0 & 0 & \cdots & 0 \\ w_{21}^{(1)} & w_{22}^{(1)} & \cdots & w_{2,S}^{(1)} & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & & & & \\ w_{H_1,1}^{(1)} & w_{H_1,2}^{(1)} & \cdots & w_{H_1,S}^{(1)} & & & & \\ 0 & & & & w_{11}^{(2)} & w_{12}^{(2)} & \cdots & w_{1,H_1}^{(2)} \\ & & & & w_{21}^{(2)} & w_{22}^{(2)} & \cdots & w_{2,H_1}^{(2)} \\ & & & & \vdots & \vdots & \ddots & \vdots \\ & & & & w_{H_2,1}^{(2)} & w_{H_2,2}^{(2)} & \cdots & w_{H_2,H_1}^{(2)} \\ \vdots & & & & 0 & & & \\ & & & & & & & \\ 0 & & & & 0 & 0 & \cdots & 0 \end{bmatrix}$$

A two layer network (one hidden layer and one output layer) of linear threshold units can be used to compute a much broader class of functions than a single layer network, as will be examined in Section 2.6. Detailed analyses of the computational power of such networks (called Perceptrons) can be found in Rosenblatt (1962) and Minsky and Papert (1968). Let us put this discussion on hold for the time being, and consider networks of linear units.

2.5 Linear Networks

While a network of LTUs with a hidden layer has tremendously more computational power than a single LTU layer, in the world of linear networks, a hidden layer makes absolutely *no difference* to the computational power of a set of hidden units. To demonstrate this, let us consider a two-layered network of linear units, with each hidden unit activity h_i computed as a weighted sum of the inputs s_1, ..., s_N, and each output unit activity computed as a weighted sum of the hidden unit activities:

$$h_j = \sum_k v_{jk} s_k$$

$$r_i = \sum_j w_{ij} h_j$$

These can be combined to form a single expression for the response values in terms of the stimulus components, as follows:

$$
\begin{aligned}
r_i &= \sum_j w_{ij} \sum_k v_{jk} s_k \\
&= \sum_j \sum_k w_{ij} v_{jk} s_k \\
&= \sum_k z_{ik} s_k
\end{aligned}
$$

where

$$z_{ik} \equiv \sum_j w_{ij} v_{jk} .$$

In vector notation, this is simply an example of the associative law of matrix multiplication: **r**=**Wh**, and **h**=**Vs** leads to **r**=**W**(**Vs**)=(**WV**)**s**=**Zs**, where **Z**≡**WV**. Thus, any function that can be computed by a two-layer network of linear units, can be computed by a one layer network (see Figure 2.5). More generally, *the functionality of <u>any</u> feedforward network of linear units can be replicated by a single layer of weights (i.e by a network with no hidden units).*

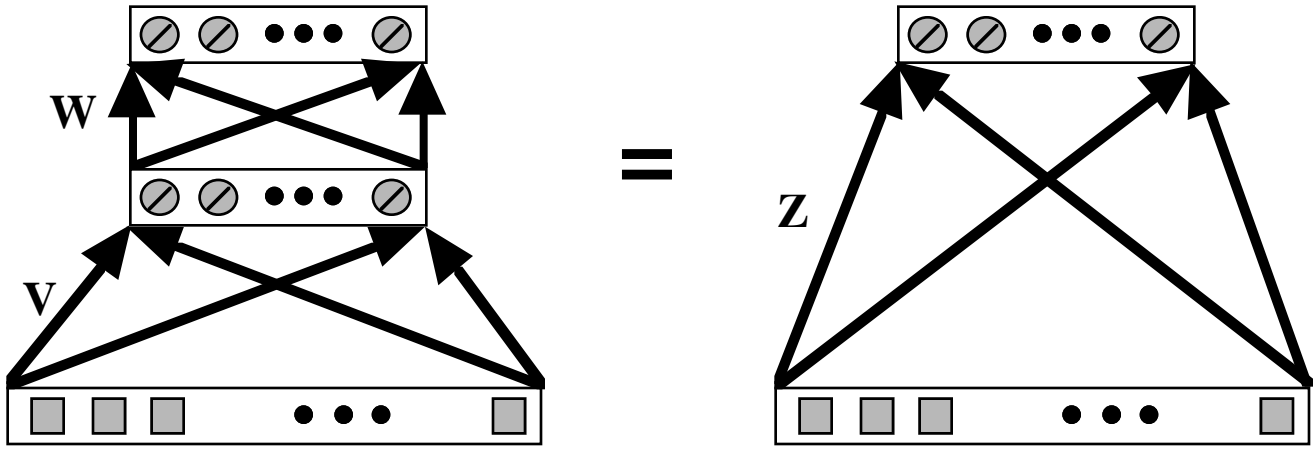
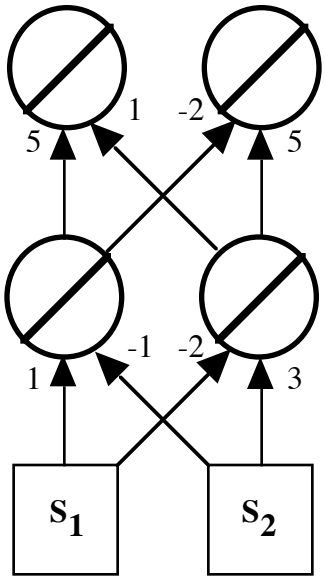


Figure 2.5. Equivalence of linear networks. Since a linear network can be reduced to a set of sums and products of matrices (linear operators), they can be reduced to a single set of weights (a single matrix, equivalent to multiplication in serial set of weights in a linear network, no matter how many hidden units, or hidden layers, are in the selected architecture.

2.6 Multi-Layered Perceptrons

Example 2.3. Find weights for a one-layer network equivalent to the network below. The lower layer and the upper layer can be expressed as weight matrices (**W** and **V**).



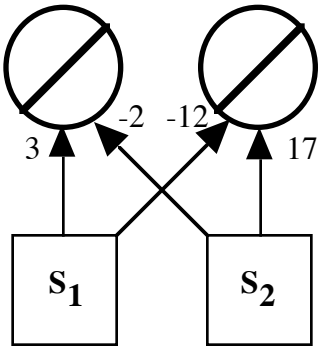
Weight matrices

$$\mathbf{W} = \begin{bmatrix} 5 & 1 \\ -2 & 5 \end{bmatrix}$$

$$\mathbf{V} = \begin{bmatrix} 1 & -1 \\ -2 & 3 \end{bmatrix}$$

The product matrix **Z** defines an equivalent one-layer matrix:

$$\mathbf{Z} = \begin{bmatrix} 5 & 1 \\ -2 & 5 \end{bmatrix} \begin{bmatrix} 1 & -1 \\ -2 & 3 \end{bmatrix} = \begin{bmatrix} 3 & -2 \\ -12 & 17 \end{bmatrix}$$



The term "perceptron", coined by Frank Rosenblatt (1958), refers to a strictly layered network of linear threshold units. Such networks are not subject to the restriction we have just demonstrated for linear networks; that is, perceptrons with multiple layers can compute functions that perceptrons with a single layer cannot. For example, consider the **XOR** task, which cannot be solved by a single layer perceptron, as shown in Section 1.2. A hidden layer with two hidden units can enable a perceptron to solve **XOR** (examine Figure 2.6). This network solution can be understood by analyzing each pattern:

- The input **(0,0)** produces a response of zero from both hidden units, giving a weighted sum of zero in the output unit, which is less than the threshold; hence $r = 0$.
- The inputs **(0,1)** and **(1,0)** each activate the **OR** hidden unit, but not the **AND** hidden unit, giving a weighted sum that is above threshold; hence $r = 1$.
- The input **(1,1)** is the only case in which the **AND** unit responds, contributing a negative influence to the weighted sum, bringing it below threshold; hence $r = 0$.

Conceptually, the output unit mimics the **OR** unit for all patterns except (1,1), since the **AND** unit is quiet for (0,0), (0,1), and (1,0). The **AND** unit supplies the "exclusion" to the **OR** (that puts the **X** in **XOR**) by responding to that single (exclusive) pattern, and having a negative weight to the output.

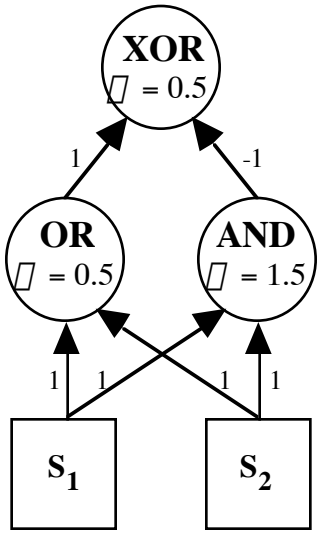


Figure 2.6. A network that computes XOR. The connections (arrows) are labeled with the corresponding weight values, and the units (circular nodes) show the function (**OR**, **AND**, **XOR**) of the inputs (square nodes) that they compute. Threshold values (θ) are shown in the unit nodes. Note that the output unit's label indicates that it computes the **XOR** of the *network inputs* (s_1 and s_2), *not* the **XOR** of its direct inputs (the hidden layer).

This particular assignment of weight and threshold parameters is not the only set by which a 2-2-1 network computes **XOR**; there are many other solutions (see Exercises). Consider what function is computed if the signs of all the weight and threshold parameters are reversed (multiplied by –1).

Here is a PASCAL implementation of a two layer perceptron:

```
procedure twolayerperceptron(patno:integer); {patno identifies one row of the
       environment array stim}

{ ************************************************************************
```

```
In this procedure, the argument patno specifies the input pattern as one row of the
environment array stim.
  ********************************************************************** }

var i, j, k : integer ;
    x : real ;  {x is the linear sum for each unit}

begin

    for i := 1 to nhid do begin
       x := hbias[i] ;
       for j := 1 to nin do x := x + v[i][j]*stim[patno][j] ;
       hidr[i] := squash(x) ;
       end ;

    for i := 1 to nout do begin
       x := rbias[i] ;
       for j := 1 to nhid do x := x + w[i][j]*hidr[j] ;
       outr[i] := squash(x) ;
       end ;

end ;
```

Now consider **PAR***n*, known for its non-linear separability: it is notoriously difficult to compute for a system of linearly separable units, since it has different values for any two inputs that differ by just one bit. A perceptron that computes the 3-bit parity task is illustrated in Figure 2.7 (as in the case of the **XOR** network above, this set of weights is not unique). Here, the processing reflects an easily articulated procedure for computing parity; namely, first count the number of **1**s in the input and determine if that number is odd or even. The hidden units in this solution effectively "count" the ones in the input as follows:

- All weights from the input variables to the hidden units have the value **1**. Thus, the weighted sum in each of these units is equal to the number of **1**s in the input.
- One of the hidden units (leftmost in the figure) responds if the weighted sum is 1 or more. Another (middle) responds if the weighted sum is two or more, and the third (right) only responds if all three input values are **1**. Thus as the number of **1**s is increased from 0 to 3, the hidden units "light up" from the left to the right; in effect, the pattern of activity of the hidden units communicates the number of 1s in the input to the output layer.
- The output unit responds if the weighted sum is greater than 0.5 (the threshold value) weight from the left hidden unit to the output unit is **1**.
- The weights from the hidden units to the output unit alternate (+1 for the weight from the left hidden unit, -1 from the middle, and +1 from the rightmost unit). Thus, the weighted sum is +1 if the number of 1s is odd, and the sum is zero if the number of **1**s is even.

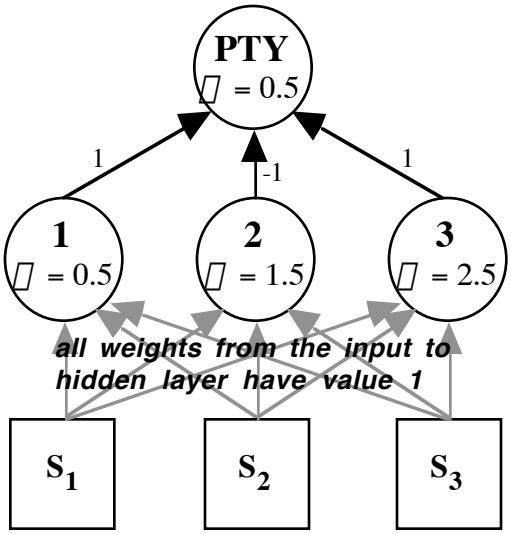


Figure 2.7. A network that computes three bit parity. All weights from the input nodes (squares) to the hidden units have the value 1. The hidden units respond according to the number of 1s in the input there are; if that number is greater than or equal to the label in a given hidden unit, that unit "fires".

The extension of this scheme to *N* bit parity computation is straightforward (see Figure 2.8). Just *N* hidden units are required, each have weights of 1, and thresholds increasing from 0.5 to *N*-0.5 in increments of 1. The reader should note that this network reduces to the XOR network of Figure 2.6 for the *N*=2 case.

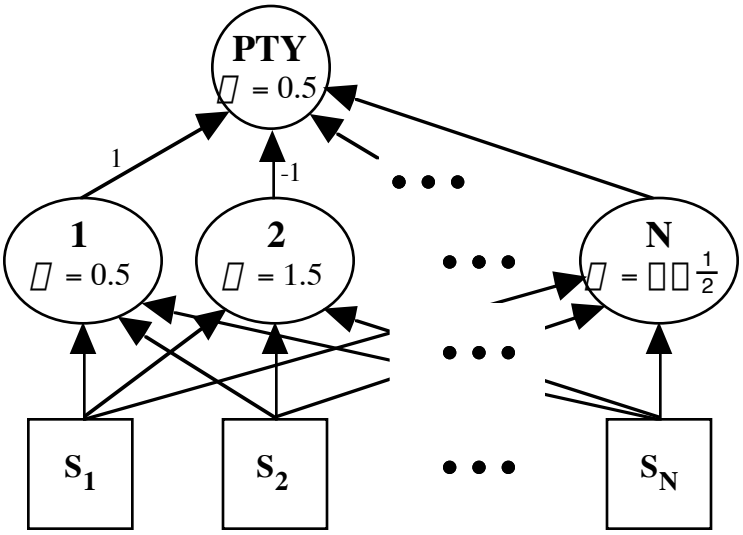


Figure 2.8. A network that computes PAR*n*. A generalization of the 3-bit parity network (previous figure). Again, all weights from the input nodes (squares) to the hidden units have the value 1. The hidden units respond according to the number of **1**s in the input there are; if that number is greater than or equal to the label in a given hidden unit, that unit "fires".

Thus, a strictly-layered network with *n* hidden units can compute **PAR***n*..

The theorem below shows that, given enough units, a Perceptron with a single hidden layer can compute any Boolean function.

> *Theorem.*: For every Boolean task **B** of *N* variables, there exists a set of weights and thresholds in a two layer perceptron (one hidden layer, and an output layer) with not more than 2^N units that will compute **B**.
>
> *Proof*: For each of the 2^N patterns on the *N* inputs, let there be a corresponding hidden unit that responds only to that pattern. Hence, for any pattern, one and only one hidden unit is active, and so the only nonzero contribution to the weighted sum in the output unit will come from the single unit activated by the input pattern; since that unit has a response value of **1**, the weighted sum is equal to the weight of the connection from the active unit to the output (Figure 2.9 illustrates this for *N*=3). Thus, for a given threshold value θ in the output unit, the units corresponding to patterns for which a **1** response is desired have a weight to the output unit that is greater than θ, the others have a weight less than θ. Note that, if θ>0, the hidden units corresponding to patterns that give a **0** response need not be included in the network at all.

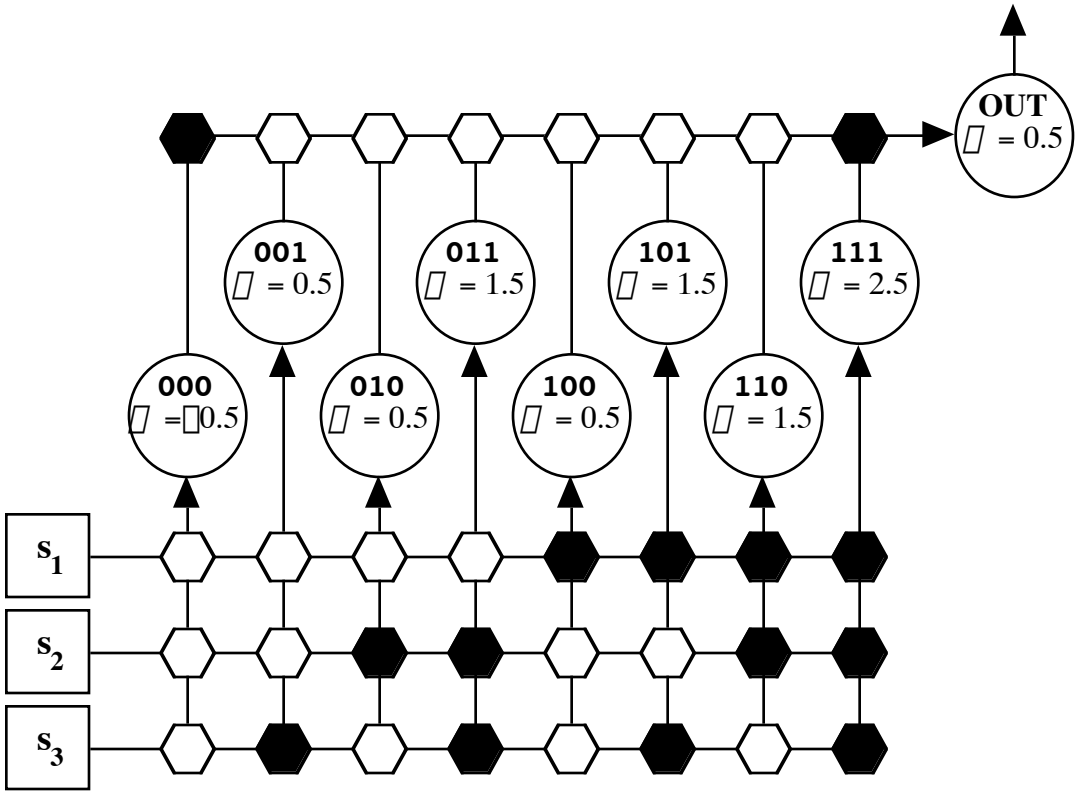


Figure 2.9. A 3-8-1 Network. The 3-by-8 array of hexagons represents weights (white=0, black=1) from the input nodes (squares) to the hidden units, which have their weights and biases configured such

that they respond to one and only one input pattern (labeled in bold on the units). Thus, for any Boolean input, exactly one hidden unit is active. The weights from the hidden layer to the output unit (upper row of hexagons) select which patterns activate the output unit. Any Boolean output function can be selected by setting the weights at this level to 1 for the hidden units corresponding to patterns for which an output of **1** is desired, and the weights to –1111 for the hidden units corresponding to patterns for which an output of **0** is desired.

A much more general theorem proven by Hornik, Stinchcombe, and White (1989) shows that multilayer feed-forward nets using arbitrary squashing functions can approximate *any* function[11] to *any* desired degree of accuracy given enough hidden units.

2.7 Representations

The pattern of activity across each layer (including the input and output layers) of a multilayer network can be viewed as a *representation* of the stimulus. The representational transformation from layer to layer enables units of limited computational power to compute functions that are *not computable* by a single layer of weights. Consider the example of a multilayered perceptron solving **XOR** as described above. The single output unit computes the **XOR** of the two input values. There is a hidden layer made up of two units, one computing the **OR** of the inputs, the other computing the **AND** of the inputs. Table 2.1 displays the response of every unit to each stimulus.

Table 2.1. Threshold Unit Solution to XOR

label	s_1	s_2	h_1 *OR*	h_2 *AND*	r *XOR*
A	0	0	0	0	0
B	0	1	1	0	1
C	1	0	1	0	1
D	1	1	1	1	0

Consider this network in light of the linear separability restriction on the individual units. That the **OR** and **AND** functions are linearly separable has been established (Section 1.1). In order to solve this task then, the representa-

[11] The theorem's validity is restricted to "Borel measureable" functions, which means functions satisfying the following condition: [condition here]. Almost any function a person would "normally" encounter satisifies this condition. It would take a malicious mathematician to dream up a function that is not Borel measureable.

tion of the patterns **A**, **B**, **C**, and **D**, that is presented to the output unit must be linearly separable. Note that the stimulus pattern to the output unit is the set of responses of the hidden units (see Figure 2.6).

Each pattern presented to the network has two other representations aside from the input representation **s**; namely the pattern of activities across the hidden units, **h**, is a *representation* of the pattern, as is the pattern (**r**) across the output units (of which there is just one, in this case). The three representations of **A**, **B**, **C**, and **D** can be read across the rows of Table 2.1 above. From the point of view of the output unit(s), the set of values on which it operates directly is the only information available as to what the stimulus is. In this case, the hidden unit representation (**h**) is what counts for the output unit. Thus, the hidden unit representation of the task must be LS. Consider the three representations of the patterns plotted in their respective spaces (Figure 2.10).

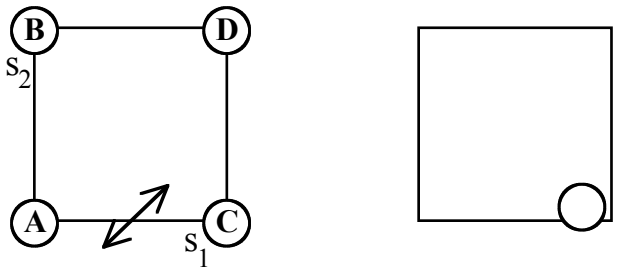


Figure 2.10. Three representations in a three layer net. The patterns **A**, **B**, **C**, **D** as they are represented at each level of the 2-2-1 perceptron. In the stimulus space (left plot), the task is not linearly separable. The two lines show the discrimination boundaries of h_1 (the **OR** function) and h_2 (**AND**). In the hidden unit space (middle), patterns **B** and **C** have the same representation (the overlapping circles are at the same coordinates), and the task is linearly separable. The response of the network to the set of patterns can also be construed as a representation, albeit one-dimensional in this case.

Local and Distributed Representations

Both neural networks and traditional AI generally function by manipulating representations of data. These take the form of real valued vectors (activity patterns) in the former, as we have seen, and symbol lists in the latter. Hence, as models of cognitive processing, these have been labeled sub-symbolic and symbolic AI, respectively (see Smolensky, 1988).

In some neural network models, patterns are distributed over several units (a *distributed* representation), while in others, each input pattern corresponds to a single unit (a *local* representation). Both approaches have roots in neurobiology. The distributed approach stems from Lashley's (1929) experiments, in which he trained rats to find their way through one of three mazes, removed a portion of their cerebral cortex, and tested each rat on the maze which it had learned. Lashley found that no particular area could be identified as the storage site for the maze

information, but that the error rate increased gradually with the amount of cortex removed and with the difficulty of the maze. This *graceful degradation* in performance led Lashley to state his Principle of Mass Action (see Box).

Lashley's Principles

Directly quoted from Lashley (1929)

Principle of Equipotentiality: "The term 'equipotentiality' I have used to designate the apparent capacity of any intact part of a functional area to carry out, with or without reduction in efficiency, the functions which are lost by destruction of the whole. This capacity varies from one area to another and with the character of the functions involved."

Principle of Mass Function: "...the efficiency of performance of an entire complex function may be reduced in proportion to the extent of brain injury within an area whose parts are not more specialized for one component of the function than another."

Experiments in the 1960s and 1970s indicated that individual neurons in certain visual areas of cortex respond to low order visual features, such as contrast edges and corners at specific orientations (Hubel and Wiesel, 1962). Other regions in cortex contain neurons that seem to be much more selective, responding to higher order features, such as hands and faces (Gross, Bruce, and Desimone, 1979). These results led to another view of neural representation in which the perceptual system was organized into layers that encoded a scene across features of increasing complexity. Based on the neurophysiological evidence, and on theoretical grounds, Barlow (1972) put forward his *single neuron hypothesis*, in the form of five "dogmas" (see Box).

Barlow's Dogmas

Directly quoted from Barlow (1972)

1. "To understand nervous function one needs to look at interactions at a cellular level, rather than either a more microscopic or microscopic level, because behaviour (sic) depends on the organized pattern of these intercellular interactions."
2. "The sensory system is organized to achieve as complete a representation of the sensory stimulus as possible with the minimum number of active neurons."
3. "Trigger features of sensory neurons are matched to redundant patterns of stimulation by experience as well as by developmental processes."
4. "Perception corresponds to the activity of a small selection from the very numerous high-level neurons, each of which corresponds to a pattern of external events of the order of complexity of the events symbolized by a word."
5. "High impulse frequency in such neurons corresponds to a high certainty that the trigger feature is present."

The neural network literature includes approaches spanning a broad range from very local representations to highly distributed representations. For example, the approach of Feldman and Ballard (1982) developed at the University

of Rochester is based on local representations[12], whereas the *parallel distributed processing*, or "PDP", approach developed at the University of California San Diego (McClelland and Rumelhart, 1986) assumes distributed representations.

2.8 Notes on Implementation

For those readers that are more at home with computer code than mathematical notation, three types of feed-forward network are implemented in PASCAL in this section:

- A strictly layered network, with all units using the same nonlinear function.
- A network composed of "banks" of units, with full connectivity between certain banks.
- A network defined by a connectivity matrix.

Naturally, the more structured the network architecture, the more compact the computer code.

Strictly Layered Implementation

A *strictly layered* network is composed of a sequence of layers between the input and output layers. The input units are all connected to all the units in the first intermediate level, which are in turn connected to all the units in the next intermediate level, and so on, until the output units (see Figure 1.10). In this case, all the unit activities in the network can be stored in a single two-dimensional array, where the first index indicates the layer and the second indicates the unit in the layer and the bias for each unit is identified by the same two index values; likewise there is a three dimensional data structure containing all the weights, where the postsynaptic layer is indicated by the first index, the unit in the postsynaptic layer is indicated by the second index, and the unit in the presynaptic layer is indicated by the third index value.

```
procedure transfer(patno:integer); {patno identifies one row of the
    environment array stim}

{ ********************************************************************
In this procedure, the argument patno specifies the input pattern as one row of the
environment array stim.  Each integer n[i] is the number of units in layer i where
i=0 is the input layer, and nlevels is the number of levels (not counting the input
```

[12] The term *connectionism* coined in this paper (Feldman & Ballard, 1982), originally referred to an approach to modeling with local representations (i.e. one unit active at a time), but has since become a more general label. In many circles, *any* neural network model could be called a connectionist model.

```
layer).  The weight value w[i][j][k] corresponds to the connection from unit k in
layer i-1 to unit j in layer i.  The response of unit j in layer i is r[i][j].  A
local variable, x holds the linear sum for each unit until the response is computed.
  ********************************************************************** }

var i, j, k : integer ;
    x : real ;  {x is the linear sum for each unit}

begin
    for i := 1 to n[0] do r[0][i] := stim[patno][i] ;
    for i := 1 to nlevels do
       for j := 1 to n[i] do begin
         x := bias[i][j] ;
         for k := 1 to n[i-1] do x := x + w[i][j][k]*r[i-1][k] ;
         r[i][j] := squash(x) ;
         end ;
end ;
```

Bank Structure

Networks that are not strictly layered can have considerable structure nevertheless; if the units are organized into groups, such that there is full connectivity between certain pairs of groups (let these groups be called *banks* of units), separate array variables can be defined to represent the the connection matrices between the banks (see Figure 2.11).

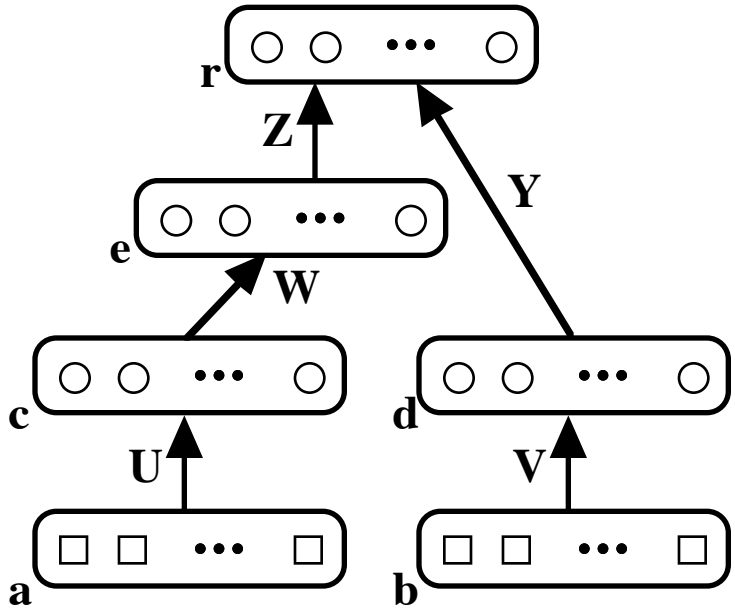


Figure 2.11. A partially layered architecture. Several banks of units and input components (lower case labels) are interconnected by weight matrices (upper case), where a bank is defined as a set of units that all receive input from a common set of units and send output to a common set.

Pascal code is shown below for the above architecture. Note that each layer's response is computed by a separate loop through the units in that layer, and that each loop begins by initializing the weighted sum to the unit bias value, then adding activity from other layers, each with a separate for-loop.

```
procedure bank_architecture(patno:integer); {patno identifies one row of the
                                             environment arrays a and b}

{ ***************************************************************************
In this procedure, the argument patno specifies the input pattern as a row in each
of the environment arrays a and b.  For each bank X (X = a, b, c, d, e, f, r), the
integer nX specifies the number of units in the bank, the linear sum in the i-th unit
is Xx[i], and the response of the i-th unit, X[i], is computed as Xfun(Xx[i]). The
weight matrices u, v, w, x, y, z are separately defined.
  *************************************************************************** }

var i, j, k : integer ;
    x : real ;  {x is the linear sum for each unit}


for i:=1 to nc do begin
  cx[i]:=cbias[i] ;
  for j:=1 to na do cx[i]:=cx[i]+u[i,j]*a[patno,j];
  c[i]:=cfun(cx[i]) ;
  end ;
for i:=1 to ne do begin
  ex[i]:=ebias[i] ;
  for j:=1 to nc do ex[i]:=ex[i]+w[i,j]*c[j];
  e[i]:=efun(ex[i]) ;
  end ;
for i:=1 to nd do begin
  dx[i]:=dbias[i] ;
  for j:=1 to nb do dx[i]:=dx[i]+v[i,j]*b[patno,j];
  d[i]:=dfun(dx[i]) ;
  end ;
for i:=1 to nr do begin
  rx[i]:=rbias[i] ;
  for j:=1 to nd do rx[i]:=rx[i]+y[i,j]*d[j];
  for j:=1 to ne do rx[i]:=rx[i]+z[i,j]*e[j];
  r[i]:=rfun(rx[i]) ;
  end ;
```

Arbitrary Structure

The connection matrix **C** can be used to describe an arbitrary network topology, as defined at the beginning of the section. Assume the units are ordered in feed-forward fashion such that there is no connection from unit j to unit i if $i \leq j$; that is, $C_{ij}=0$ for all $i \leq j$. Note that the weights are subject to the same restriction ($w_{ij}=0$ for all $i \leq j$). Elements of **C** in columns associated with input values are all zero, as are the elements in rows of output units. Let N be the total number of nodes, including input nodes and output units. The program begins with a short routine that labels the units with an N-element array **node_type** (code for constructing **node_type**, given **C** is left as an exercise):

$$\texttt{node\_type[i]} \equiv \begin{cases} 0 & \text{if } i \text{ is an input node} \\ 1 & \text{if } i \text{ is a hidden unit} \\ 2 & \text{if } i \text{ is an output unit} \end{cases}$$

The **`activity[i]`** values are then computed for each unit in order, by first copying the value of the stimulus vector into the components of activity corresponding to input nodes.

```
if (node_type[i]=0) then activity[i]:=stim[i] ;
```

The i-th computational unit is a function of the units with indices $j < i$, for which $C_{ij} \neq 0$.

```
procedure arb_architecture(patno:integer); {patno identifies one row of the
                                              environment array stim}

{ ***************************************************************************
In this procedure, the argument patno specifies the input pattern as a row in the
environment arrays stim.  wtsum is a local variable that holds the linaer sum and is
used to compute the activity of node i.

  *************************************************************************** }
var i, j, k : integer ;
    wtsum : real ;  {wtsum is the linear sum for each unit}

if (node_type[i]=0) then activity[i]:=stim[patno,i] ;
if (node_type[i]>0) then begin
  wtsum:=bias[i] ;
  for j:=1 to i-1 do if (C[i,j]>0) wtsum:=wtsum+w[i,j]*activity[j] ;
  activity[i]:=squash(wtsum);
  end;
```

2.9 Exercises

1. Let **C** be the connectivity matrix of an acyclic network. Show that reversing the direction of all the links results in a network that is also acyclic. Let **D** be the connectivity matrix of the reversed network, and find an expression for the elements of **D** in terms of the elements of **C**.

2. Draw networks corresponding to each of the three connectivity matrices $\mathbf{C}_1$, $\mathbf{C}_2$, and $\mathbf{C}_3$. Which of the following connectivity matrices correspond to acyclic networks? Which of these are strictly layered?

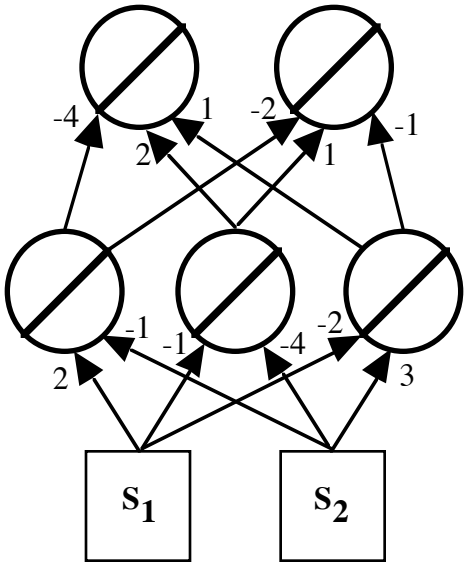
$$\mathbf{C_1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix} \quad \mathbf{C_2} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \quad \mathbf{C_3} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

3. Find a weight matrix and a set (vector) of thresholds for a layer of linear threshold units that compute the following 2-by-3 mapping:

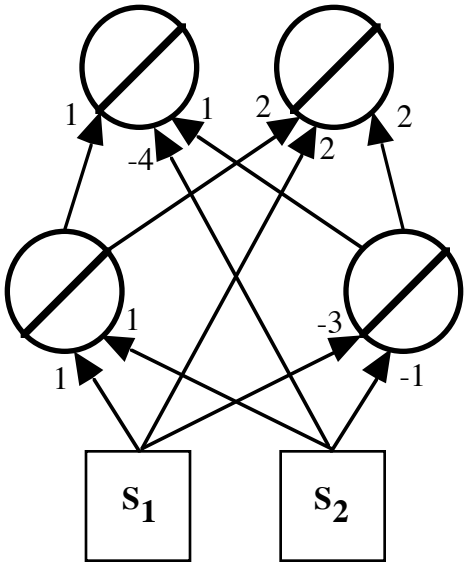
Stimulus	Target
0 0	0 0 1
0 1	0 0 0
1 0	0 1 0
1 1	0 0 0

4. For each the following *linear* networks, find a set of equivalent weight values:

a.



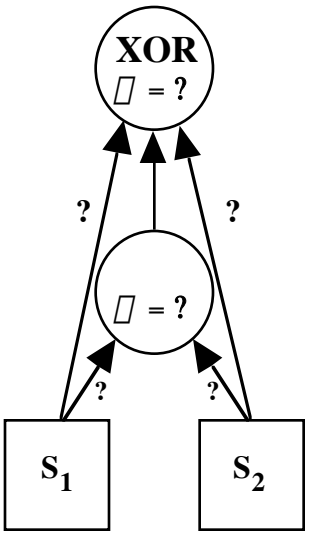
b.



5. Find an alternative set of weights and thresholds for the **XOR** task two hidden units that respectively compute the **NAND** and **NOR** functions (rather than the **AND** and **OR** in the example).

6. The hidden unit representations that result from having **AND** and **OR** hidden units have been shown to enable computation of **XOR**. Which, if any, of the functions that are linearly separable in their input representations are no longer linearly separable when mapped to the (**AND**,**OR**) representation of the hidden layer?

7. Find weights and thresholds that will compute the **XOR** task using just one hidden unit plus direct connections from the input to output layers. [HINT: note that the input into the **XOR** unit now has three components -- find a LS function for the hidden unit, such that the 3-bit task into the output unit is LS]



8. Write a computer routine to compute the output of a feed forward network like the one above, but with an arbitrary number of hidden units and an arbitrary number of output units. Or, put another way, the program should simulate a network that has one hidden layer, with input-hidden connections, hidden-output connections, *and* direct input-output connections.

9. Write a computer program that constructs the array **node type**, given a sorted connectivity matrix **C**.

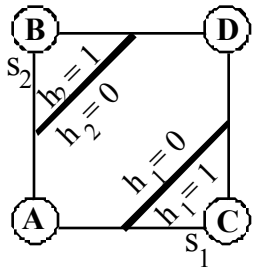
10. Write a computer routine to compute the response of a strictly layered network with *two* hidden layers.

11. Show that the task below (3 bit **XOR**) can be computed by a Perceptron with a single hidden layer of two hidden units. [HINT: let one hidden unit compute a 3 bit "OR", and the other compute a 3 bit "AND"]

Item	Stimulus	Response	Item	Stimulus	Response
1	0 0 0	0	5	1 0 0	1
2	0 0 1	1	6	1 0 1	1
3	0 1 0	1	7	1 1 0	1
4	0 1 1	1	8	1 1 1	0

12. Write a computer routine that generates a vector **`n_inputs`**, given a connection matrix **C**, where each component **`n_inputs[i]`** is the number of nodes that supply input values for unit *i*.

13. Find weights and thresholds for units h_1 and h_2 in the *S*-space diagram below. Sketch the representations in *H*-space as in Figure 2.10. Find weights and thresholds for an LTU that computes **XOR**(s_1, s_2) using h_1 and h_2 as inputs.



S Space

3. Generalization, Curve Fitting, and Regression

Let us define *generalization* as the ability to extract statistical regularities from a set of input-output data, such that a system can respond appropriately to input stimuli even (and especially) for stimuli not in the original data set. Throughout the modern era, generalization of this kind has been the domain of statisticians, who have developed several approaches, of which the most well known is called *regression*, perhaps better known as *curve fitting*. By way of introduction or review (depending on the reader), a brief overview of linear regression follows in the next section. Since real-world data values show some degree of randomness, a primary difficulty with extracting relations from data is that it is not clear how to separate the underlying regularities (the "actual" relationships) between the data variables and spurious relationships idiosyncratic to the particular data sample.

Statisticians are often faced with the chore of finding systematic relationships in sets of data. Generally, the approach is to assume that the data follows some particular type of *parametric* relationship, and then to find those parameters that best account for the data; this is called *fitting the data*. If the data variables are strongly related, it should be possible to *predict* the value of some variables, given the values of others.

Assume there are a set of data items, having several variables that are thought to be related. For example, each data item might be several measurable variables of a house: the size in square feet, the number of bedrooms, whether or not it has central air conditioning, and its fair market value (see Table 3.1). Given enough experience, a trained real estate agent in a given region can predict the last variable (value) given the first three. The prediction cannot be perfectly accurate however because of missing information, like the age of the house, its condition, the size of the lot, the number of bathrooms, the neighborhood and so on. These missing variable values cause certain data to *seem* inconsistent; for example items 3 and 6 both have 2 bedrooms and AC, only differing in square footage, yet the larger house has a lower value. Most real world problems are not 100% predictable due to so-called "noise" in the data, often attributed to "random fluctuations" (in fact, these can often be attributed to missing information).

Table 3.1. Predict the price of house #9 given the prices on 1-8.

House ID	Sq Ft	Bedrooms	AC	Value
1	1200	2	no	$90000
2	1500	3	no	$125000
3	1500	2	yes	$115000
4	1700	3	yes	$120000
5	1800	4	no	$170000
6	1000	2	yes	$160000
7	3000	4	yes	$280000
8	2200	4	no	$210000
9	1800	2	yes	**?????**

3.1. Linear Regression and the Error Surface

Given a set of data that is thought to have a roughly linear relationship, the technique of linear regression can be used to find the constants that specify the relationship. Let the data be of the form (x_i, y_i) for N data items (i=1,...,N); for example, suppose x_i is the height and y_i is the weight of person i, and we have data for N individuals. Linear regression rests on the assumption of a linear relationship between x and y; that is, we want to find parameters m and b that satisfy, *as well as possible*, the relationship $y_i = mx_i+b$ for all i. In general, real-world data (as opposed to idealized synthetic, or "toy" data) will not conform exactly to a linear relationship, or any pre-specified function. Having assumed a linear function approximates the dependence of y on x, the first step in regression is to introduce a measure of "fitness" of any given line to the data. That is, *which line* (as specified by m and b) best account for the given data set? While, there are infinitely many ways to define the measure of the fitness of a line, the most common definition is to take the values of y that would correspond to each x_i in the data set and compare them to the actual y_i values from the data; that is, y_i is compared to $mx_i + b$ for each data item (for all i). If y_i tends to be close to $mx_i + b$, then m and b specify a line that is a "good fit" to the data. So we define a measure of overall error E as follows:

$$E = \sum_{i=1}^{N} (y_i - f(x_i))^2 = \sum_{i=1}^{N} (y_i - (mx_i + b))^2 \qquad [3\text{-}1]$$

Example 3.1. Find the slope and intercept of the line that gives an optimal least squares fit to the following set of *x*-*y* coordinates: (0,1), (1,3), (2,1), (3,2).

Begin by creating a table with a row for each item, with columns corresponding to x, y, x^2, and xy, summing each column to give the coefficients:

x values	y values	x^2	xy
0	1	0	0
1	3	1	3
2	1	4	2
3	2	9	6
6	7	14	11

The values in the last row can simply be substituted into the formula:

$$m = \frac{N\sum_{i=1}^{N} x_i y_i - \left(\sum_{i=1}^{N} x_i\right)\left(\sum_{i=1}^{N} y_i\right)}{N\sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{N} x_i\right)^2} = \frac{(4)(11) - (6)(7)}{(4)(14) - 36} = 0.1$$

$$b = \frac{\left(\sum_{i=1}^{N} x_i^2\right)\left(\sum_{i=1}^{N} y_i\right) - \left(\sum_{i=1}^{N} x_i\right)\left(\sum_{i=1}^{N} x_i y_i\right)}{N\sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{N} x_i\right)^2} = \frac{(14)(7) - (6)(11)}{(4)(14) - 36} = 1.6$$

The i-th term in the sum is a measure of the discrepancy between the line's estimate of the y value that corresponds to x_i (see Figure 3.1). Note that the terms in the sum are the *squares* of the differences between y_i and $mx_i + b$, so that no term can decrease the error.

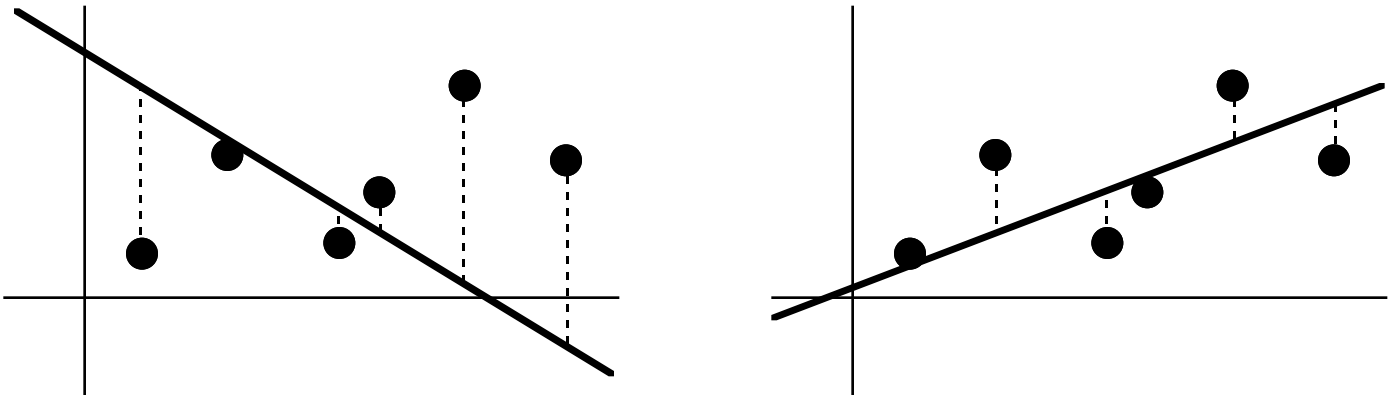


Figure 3.1. Looking for linear relationships in data. The bold line in each figure is a possible "fit" to the given data points (dark circles). The dashed lines indicate the deviations of the points from the proposed lines. Since these deviations are smaller in the figure on the right, it is considered a better fit. Thus, the fitness of a line can be quantified, and subsequently optimized.

The difference between y_i and the value of y computed by the linear function ($mx_i + b$) is squared for each term in the summation so that each term in the sum is positive, and so positive and negative differences between y_i and $mx_i + b$ cannot cancel one another out. Let us define the best fitting line to a given set of data, as that line for which E is a minimum. Of course, there are alternative error measures, for which the optimal line may be different. Some of these will be discussed in Section 4.4. Thus, the best fit to the data can be found by determining the values of m and b that minimize E. Assuming E has a minimum (which is a correct assumption, it turns out), the partial derivatives of E with respect to each of the two parameters (m and b), are zero at that point.

The partial derivatives can be computed directly using the chain rule from Eq [3-1], but expanding the expression into a form that is quadratic in m and b (i.e. of the form $a_1b^2+a_2mb+a_3m^2+a_4b+a_5m+a_6$), where the a_i are constants) supports our assumption (hope) that there are values of m and b that minimize E.

$$E = Nb^2 + m^2 \sum_{i=1}^{N} x_i^2 + 2mb \sum_{i=1}^{N} x_i - 2b \sum_{i=1}^{N} y_i - 2m \sum_{i=1}^{N} x_i y_i + \sum_{i=1}^{N} y_i^2 \qquad [3\text{-}2]$$

The partial derivatives of E in terms of the quadratic expansion can now be computed in terms of the coefficients (a_i), each of which is a constant for a given set of data.

$$\begin{aligned} \frac{\partial E}{\partial m} &= a_2 b + 2a_3 m + a_5 = 2b \sum_{i=1}^{N} x_i + 2m \sum_{i=1}^{N} x_i^2 - 2 \sum_{i=1}^{N} x_i y_i \\ \frac{\partial E}{\partial b} &= 2a_1 b + a_2 m + a_4 = 2Nb + 2m \sum_{i=1}^{N} x_i - 2 \sum_{i=1}^{N} y_i \end{aligned} \qquad [3\text{-}3]$$

Setting the expressions for both partial derivatives to zero gives a pair of linear equations, the simultaneous solution of which gives a unique (*m*,*b*) pair where the error surface is flat, indicating a possible minimum, maximum, or "saddle point". Since the second derivatives of *E* with respect to both *m* and *b* are positive (this can be simply verified by taking the derivatives of the above equations), the point (*m*,*b*) is a minimum.

$$\begin{aligned} \left(\sum_{i=1}^{N} x_i^2\right) m + \left(\sum_{i=1}^{N} x_i\right) b &= \sum_{i=1}^{N} x_i y_i \\ \left(\sum_{i=1}^{N} x_i\right) m + N\, b &= \sum_{i=1}^{N} y_i \end{aligned} \qquad [3\text{-}4]$$

Solving the two equations for *m* and *b* gives[13]:

$$m = \frac{N\sum_{i=1}^{N} x_i y_i - \left(\sum_{i=1}^{N} x_i\right)\left(\sum_{i=1}^{N} y_i\right)}{N\sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{N} x_i\right)^2} \qquad b = \frac{\left(\sum_{i=1}^{N} x_i^2\right)\left(\sum_{i=1}^{N} y_i\right) - \left(\sum_{i=1}^{N} x_i\right)\left(\sum_{i=1}^{N} x_i y_i\right)}{N\sum_{i=1}^{N} x_i^2 - \left(\sum_{i=1}^{N} x_i\right)^2}$$

3.2. Determining Weights of a Linear Unit from Examples

Regression can also be performed on a linear unit to determine the set of weights that is optimal with respect to a set of data. Given a set of *target responses* to a set of hypothetical stimuli, a set of weights can be found for a linear unit that will have a minimum error. From [earlier], the transfer function of a linear unit is expressed as

$$r = w_1 s_1 + w_2 s_2 + \cdots + w_N s_N = \sum_{i=1}^{N} w_i s_i = \mathbf{w} \bullet \mathbf{s} \qquad [3\text{-}5]$$

Let the given set of data be a set of stimulus-target pairs (s^{μ},T^{μ}); that is, for a given data item μ, T^{μ} is the "desired" response to the stimulus $\mathbf{s}^{\mu}$, where the superscript denotes which the item, and the subscript is reserved to denote the vector component of the stimulus; i.e. $\mathbf{s}^{\mu} = (s_1^{\mu}, s_2^{\mu}, \ldots, s_N^{\mu})$, where *N* is the number of input variables. As in the regression derivation above, let the error on a given item, E^{μ}, for a given set of weights be defined as the square

[13] The linear system $\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned}$ has the following general solution for *x* and *y*: $x = \frac{ed - bf}{ad - bc}$, $y = \frac{af - ce}{ad - bc}$

of the difference between the function value ($r^{\square}=w\bullet\mathbf{s}^{\square}$) and the target value for that item, $T^{\square}$. The total error over the entire data set is then:

$$E = \sum_{\square=1}^{K} \left(T^{\square} - \sum_{i=1}^{N} w_i s_i^{\square} \right)^2 \qquad [3\text{-}6]$$

where K is the number of stimulus-target pairs in the data set.

$$\frac{\partial E}{\partial w_i} = 2 \sum_{\square=1}^{P} \left(-s_i^{\square} \right) \left(T^{\square} - \sum_{k=1}^{N} w_k s_k^{\square} \right) = 0 \quad \text{for } i = 1 \ldots N \qquad [3\text{-}7]$$

Thus, for this case of minimizing a quadratic error function on a linear unit, the result is a set of linear equations.

$$\sum_{\square=1}^{P} \sum_{k=1}^{N} s_i^{\square} s_k^{\square} w_k = \sum_{\square=1}^{P} s_i^{\square} T^{\square} \quad \text{for } i = 1 \ldots N \qquad [3\text{-}8]$$

From linear algebra, we know that if the existence of solutions depends on the independence of the equations, the number of variables, and the number of equations. In this case, N equations are to be solved for the N weight parameters. Consider Example 3.2.

Example 3.2. Find a set of weights that give the lowest summed squared error to the following three stimulus-target pairs (where $\square$ = 1, 2, 3, is used to label the pairs).

$\square$	Stimulus	Target
1	−1 1	2
2	2 1	5
3	0 1	4

Plugging the data into Eq. [3-8] produces the following system. The tables detail the construction of each equation term by term.

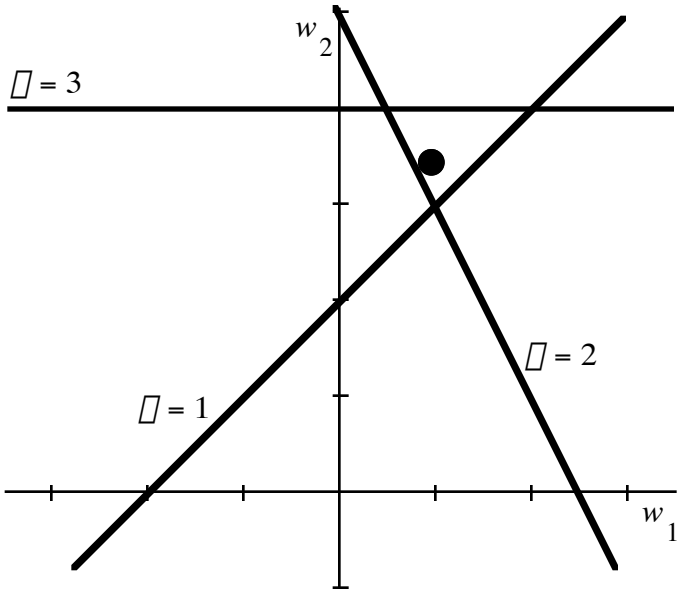
i = 1	$\sum_{\square=1}^{P} \sum_{k=1}^{N} s_i^{\square} s_k^{\square} w_k$		$\sum_{\square=1}^{P} s_i^{\square} T^{\square}$
	k = 1	k = 2	
$\square$ = 1	(−1)(−1)w_1	(−1)(+1)w_2	(−1)(+2)
$\square$ = 2	(+2)(+2)w_1	(+2)(+1)w_2	(+2)(+5)
$\square$ = 3	(0)(0)w_1	(0)(+1)w_2	(0)(+4)
$\sum_{\square}$	5w_1	+w_2	= 8

$i = 2$	$\sum_{\mu=1}^{P}\sum_{k=1}^{N} s_i^\mu s_k^\mu w_k$		$\sum_{\mu=1}^{P} s_i^\mu T^\mu$
	$k = 1$	$k = 2$	
$\mu = 1$	$(+1)(-1)w_1$	$(+1)(+1)w_2$	$(+1)(+2)$
$\mu = 2$	$(+1)(+2)w_1$	$(+1)(+1)w_2$	$(+1)(+5)$
$\mu = 3$	$(+1)(\ 0)w_1$	$(+1)(+1)w_2$	$(+1)(+4)$
$\sum_\mu$	w_1	$+3w_2$	$= 11$

The two equations (bottom rows of the tables) are easily solved:

$$\begin{aligned} 5w_1 + w_2 &= 8 \\ w_1 + 3w_2 &= 11 \end{aligned} \quad \Rightarrow \quad w_1 = \frac{13}{14} \quad \text{and} \quad w_2 = \frac{47}{14}$$

Note that this solution generates a nonzero error for each of the three stimuli. No zero-error solution exists, since the three nullclines corresponding to the given data do not intersect at a single point; the three nullclines (bold) and the minimum error weight state (dot) are drawn in the figure below.



A similar approach can be used to find the optimal parameters for a function of any form. Unfortunately, unlike the case of linear regression, the derived set of simultaneous equations is not linear for all types of parametric functions, or even solvable in closed form.

3.3. Nonlinear Regression

What about variables related by functions that are not linear? The general approach of setting derivatives to zero and solving the resulting equations can be applied to a polynomial function of any order, N; that is any function of the form

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_N x^N .$$

The derivatives of the summed square error with respect to the parameters a_i give N equations in N variables, which (as we know from linear algebra) almost always have a unique solution. The details of the solution of this system will not be discussed here (the N=2 case is the subject of exercise 3 at the end of this chapter).

It is common at this point for an astute student to observe that a sufficiently high order polynomial can provide an *exact fit* to a data set; we just have to choose N to be K-1, where K is the number of data points. For example, a straight line (N=1) can be exactly fit to 2 data points, a parabola (N=2) can be found that intersects 3 arbitrary points, etc. Figure 3.2 shows the fit of a quartic (N=4) curve to a set of 5 data points.

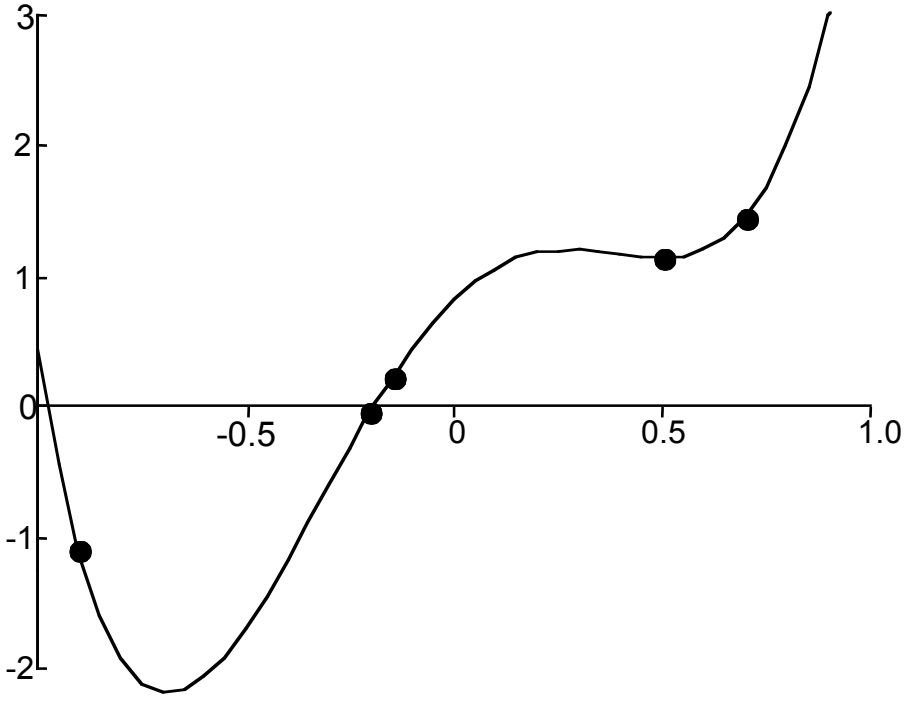


Figure 3.2. Fitting a polynomial. Five x-y points (circles) that are nearly but not exactly linearly related. The points can be exactly fit by a 4th order polynomial.

Functions other than polynomials can be fit to data also. We simply need to solve the set of simultaneous equations that result when the derivatives of the cost function with respect to all the parameters are set to zero.

3.4 Overfitting

However, the exact fit of a high order polynomial is problematic, in that it tends to generalize poorly (see Figure 3.3). That is, *y*-estimates for certain *x* values seem much less plausible for the higher order polynomial even though it fits the given data more exactly. As a rule, functions with too many parameters will overfit the data, and functions with too few parameters will underfit the data. The trick is to have neither too many nor too few parameters in the function. The neural network functions we will be using in later chapters tend too have many more parameters than necessary, but we will also examine methods to prevent overfitting.

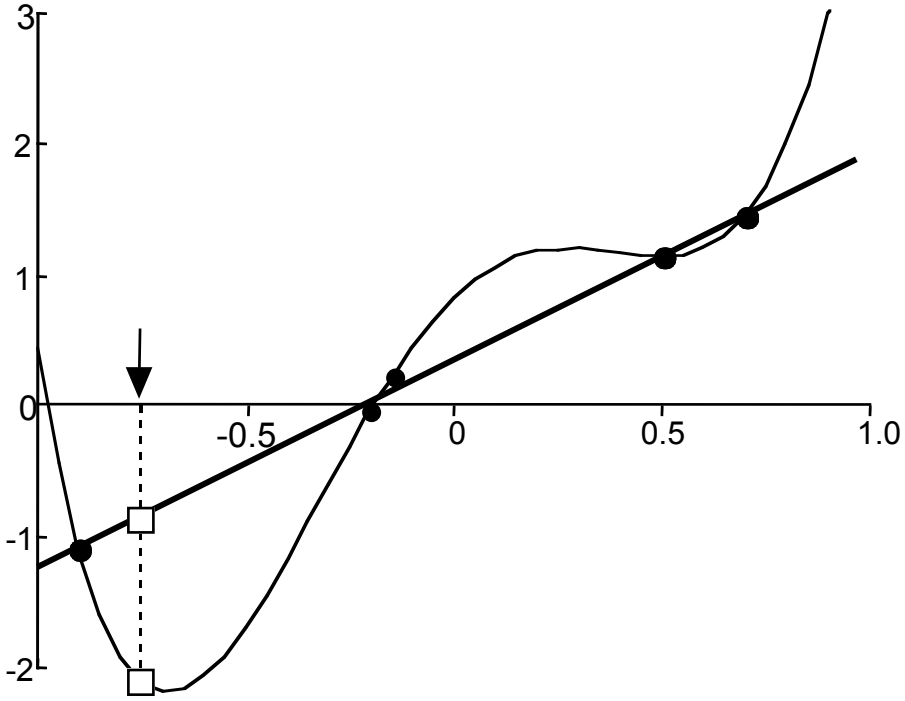


Figure 3.3. Different polynomials fit to the same data. A linear fit (bold line) to the same data points from Figure 3.2 is superimposed on the exact 4-th order fit (thin line). Consider the *y*-value predictions made by the two functions for *x*=-0.75 (arrow and dashed line).

Data from the so-called "real world" does not, in general, conform to an exact mathematical formula; that is, there are usually influences from variables that are not considered, or random (whatever that means) effects. It is often difficult to ascertain whether relationships between variables in data are due to functional relationships ("real") or artifacts of the particular data sample ("coincidental").

In general, the more parameters that can be adjusted in a function, the more versatile the function (i.e., the closer it can come to the data points); given enough parameters, a curve can exactly fit the data. However, in spite of what our intuitions might suggest, an exact fit is generally *not* desirable (see Figure above). In order to test general-

ization, an error measure is computed over a subset of the data that is withheld from the regression process. That is, a given set of data $\mathcal{D}$ is partitioned into two subsets, $\mathcal{D}^{Tr}$ for training and $\mathcal{D}^{Ts}$ for testing generalization. The regression parameters are fit to the former set, leaving the latter (testing) set to be used to evaluate both the *type* of function used to model the data, and the parameters determined by the regression. Of course, the test set is assumed to have no systematic properties that distinguish it from the training set. Ideally, both sets are randomly drawn from the same sampling process. In practice, however, the regression parameters always depend to some (hopefully small) degree on the idiosyncrasies of the data sample.

3.5. Exercises

1. For each of the following sets of stimulus-response pairs, find the weights that minimize the error. Plot the nullclines and optimal weights.

a

Stimulus	Target
1 2	2
2 -1	4

b

Stimulus	Target
3 1	-2
-1 2	3
2 3	0

c

Stimulus	Target
-2 4	2
3 -6	0

2. Repeat exercise 1b, substituting double the values for the stimulus components and the target value for the first **s**-T pair only; that is, use **s** = (6,2) and T = -4. Note that the nullcline corresponding to this pair is the same. How does this substitution influence the values of the optimal weights? What can you conclude about the relationship between the optimal weight state and the nullclines?

3. Find the gradients of the summed squared error with respect to the parameters for a quadratic function ($y = ax^2 + bx + c$). Show that setting these gradients to zero gives three linear equations for a, b, and c.

4. A parametric equation of the form $y = \alpha e^{\beta x}$ can be fit to data using linear regression by converting it to a linear equation (take the logarithm of both sides of the equation). Find α and β that fit the following x-y pairs: (0,1), (1,2), (2,5), (3,8), (4,15).

5. The deviation of linear regression in Section 3.1 is based on setting derivatives of the error to zero and solving the resulting equations. Show that the resulting parameters *minimize* the error (rather than maximizing it).

4. Gradient Optimization

The technique of solving a set of simultaneous equations (determined by setting the derivatives to zero) is not generally viable if those simultaneous equations are not linear in the parameters. In Part II, it will be seen that nonlinear transfer functions are crucial, and so it will not be possible to determine the error-minimizing parameters nearly as easily. An alternative technique is to start with an initial "guess" for the parameter values, to evaluate its error, then to gradually update the parameter values. If this is done according to certain principles, the parameter values will reduce the error incrementally, ultimately converging to a set of parameters that will hopefully (but not necessarily) be optimal. Thus, rather than computing the parameters in a single step ('boom!"), for more complex functions we must result to a slower *process*, which will show interesting correspondences with the time-course of actual *learning processes*. Thus the notion of *gradient optimization*, is introduced in the next chapter, and will be used in Chapters 6 and 7 as an alternative approach to finding a set of parameters for a function that minimizes the error for a neural network. Before discussing gradient optimization, some preliminary concepts from *dynamical systems theory* are presented.

4.1. Dynamical Systems

Immediately upon its discovery/invention and ever since, differential calculus has been used as a language for describing physical laws. The interrelationship between a set of measurable quantities in the world (x_1, x_2, ...) can often be usefully expressed in terms of the changes over time on each variable that is imposed by the other variables using a system of differential equations as follows:

$$\begin{aligned}
\frac{dx_1}{dt} &= f_1(x_1, x_2, \ldots, x_n) \\
\frac{dx_2}{dt} &= f_2(x_1, x_2, \ldots, x_n) \\
&\vdots \\
\frac{dx_n}{dt} &= f_n(x_1, x_2, \ldots, x_n)
\end{aligned},$$

or more succinctly as

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x}) \qquad [4\text{-}1]$$

Since derivatives with respect to time are so prevalent in dynamical systems, let us adopt the standard convention of denoting the time derivative with respect to a variable by a dot over that variable:

$$\text{First time derivative}: \dot{x} \equiv \frac{dx}{dt}$$

$$\text{Second time derivative}: \ddot{x} \equiv \frac{d^2x}{dt^2}$$

$$\text{Third time derivative}: \dddot{x} \equiv \frac{d^3x}{dt^3}$$

Thus, the dynamical system above can be written

$$\dot{x}_j = f_j(x_1, x_2, \ldots, x_n), \text{ for } j = 1 \ldots n$$

A set of equations in this form is called a **homogeneous** (since there is no explicit time-dependence) **first order** (since the time derivatives are *first* derivatives) **dynamical system**. A great deal is known about the formal properties of such systems, but here we will just acquaint ourselves with a few terms and concepts. More importantly, perhaps, this chapter presents techniques for how to express dynamical systems as computer routines that can be used to numerically simulate their behavior. The reader should be aware that the treatment here is minimal; additional reading[14] or coursework in dynamical systems will give the reader deeper insight into many neural network learning procedures (including backprop).

[14] Some recommended texts are Boyce and DiPrima (1996) and Davis (1992).

The State of a First Order System

The vector of values, $\mathbf{x}(t) = (x_1(t), ..., x_n(t))$, at time t is the *state* of the system at that time. In terms of the state vector, the dynamical system can be expressed $\dot{\mathbf{x}} = f(\mathbf{x})$. Casting the state as a vector leads to a spatial conceptualization (as we have already seen in our plots of weight space for example), in which the similarity of one state to another can be measured using vector operators, such as the inner product. In order to use a dynamical system to predict the state over time the state vector must be specified for some initial time, t_0. From this *initial state* $\mathbf{x}(t_0)$, the differential equations indicate subsequent states $\mathbf{x}(t)$; $t > t_0$. The change in state determined by the dynamical system for each state vector $\mathbf{x}$ is also a vector, $\dot{\mathbf{x}} \equiv (\dot{x}_1, \dot{x}_2, ..., \dot{x}_n)$. Thus, an image of the dynamical system can be constructed by placing a change vector $\dot{x}$ at each point in the state space (see Figure 4.1).

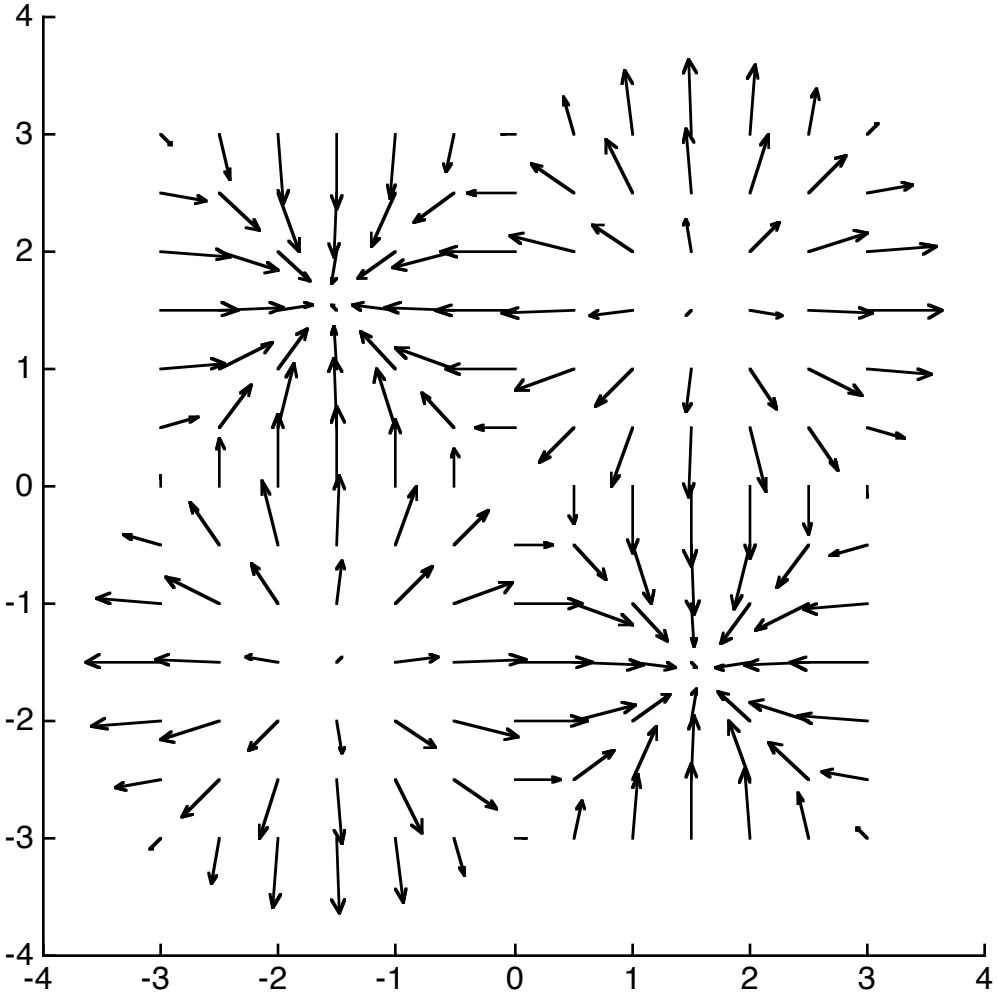


Figure 4.1. Plotting a first order system in two dimensions. Changes in the state (x,y) can be represented by positioning an arrow with its tail at (x,y). Here, the system $\dot{x} = -(\cos x)(\sin y)$, $\dot{y} = -(\sin x)(\cos y)$ is plotted in the x-y plane.

Trajectories

From the initial state $\mathbf{x}(t_0)$, $\mathbf{x}$ changes in a continuous fashion (we assume the dynamical system function $f(\mathbf{x})$ is continuous); thus, the state traces a *trajectory* through the state space. Note that, since the change in state is purely a function of the state, trajectories from different initial points never cross one another, nor can a trajectory cross itself; otherwise, two different directions would be indicated from the point of intersection. Due to topological constraints, a trajectory for a 2-D dynamical system must either converge to a point (usually this convergence is asymptotic), diverge to infinity, or converge to a closed loop, called a limit cycle[15]. Figure 4.2 illustrates the possibilities for a 2-D system. In systems of higher dimensionality, it is possible to stay within a bounded region without converging to a point or limit cycle, but such trajectories must still exhibit oscillatory behavior of some kind, or convergence with respect to all variables.

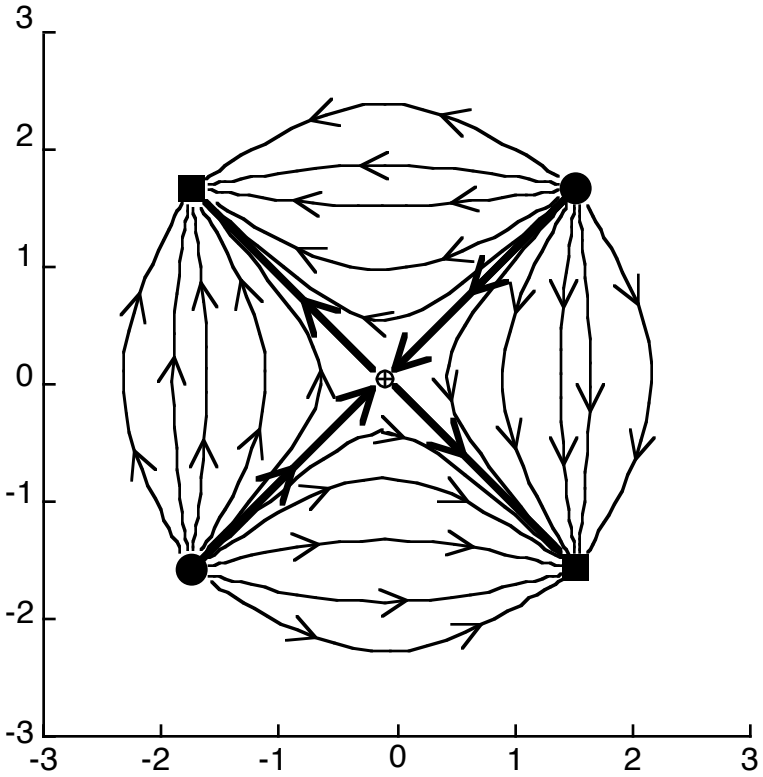


Figure 4.2. Trajectories. Trajectories are shown for the 2-D dynamical system from the previous figure. Trajectories intersect at equilibrium points, of which three types are depicted: *attractors* (filled squares), *repellors* (filled circles), and a *saddle point* (⊕).

[15] The reader is invited to experiment with pencil and paper. Try drawing non-intersecting paths that do not all either converge to a point, diverge off the page, or converge to a closed loop.

Equilibrium States and Stability

An **equilibrium state** (or **equilibrium point**) is a state for which time derivatives with respect to all variables are zero; thus, a state at equilibrium stays there. The **stability** of an equilibrium point is the tendency for a state in equilibrium to remain there if it is (for some reason) displaced slightly from that point. Some equilibria are **attractors**, in that all states within some neighborhood of an attractor (called the **attractor region**) are moved toward the attractor by the dynamical system. Such an equilibrium point is called **stable**. Other equilibria have the opposite properties; states that lie in the neighborhood of such **repellors** are driven out of the neighborhood by the dynamical system. The ultimate fate of such a trajectory depends on the dynamical system, it may diverge to infinity, or be "sucked in" to an attractor in another part or the state space, or converge to a limit cycle. Thus, while a state that lies *exactly* on an **unstable** equilibrium point, will remain there forever unless it is perturbed (slightly moved), in which case it follows a trajectory that leads it away from that point. In real systems, random perturbations are generally unavoidable, hence unstable equilibrium conditions are generally transient, while stable equilibria can persist depending on the size of the attractor region relative to the size of the perturbations.

The equilibrium state(s) of a system can be found by setting all the derivative functions (f_i(x) in Equation 4-1) equal to zero, and simultaneously solving the resulting set of "equilibrium equations". The solution set to any one of these equations is called the **nullcline** for that equation. The nullcline corresponding to a linear differential equation is thus a linear manifold (i.e., a line in 2-D, a plane in 3-D, or a hyperplane in higher dimensions). Note that the trajectories that cross the nullcline corresponding to the derivative of x are vertical; similarly, those corresponding to the derivative of y are horizontal. Also note that equilibria are located at the intersection of nullclines corresponding to all variables.

Example 4-1. Plot nullclines and trajectories for the linear system

$$\begin{aligned} \dot{x} &= -2x + y \\ \dot{y} &= -x - y \end{aligned} \qquad [4\text{-}2]$$

The nullclines correspond to the conditions $\dot{x} = 0$ and $\dot{y} = 0$; hence we merely need to plot $-2x+y=0$ and $-x-y=0$, which will obviously intersect at (0,0). Observe that on the two sides of the $\dot{x}$ nullcline (labeled $\dot{x} = 0$ in Figure 4.1), $\dot{x}$ has different signs, and likewise for the $\dot{y}$ nullcline. The two nullclines thus partition the state space into 4 regions, each labeled in the figure by a pair of arrows indicating the sign of the change in x and y for that region. All change vectors $(\dot{x}, \dot{y})$ in a given region are in the indicated quadrant.

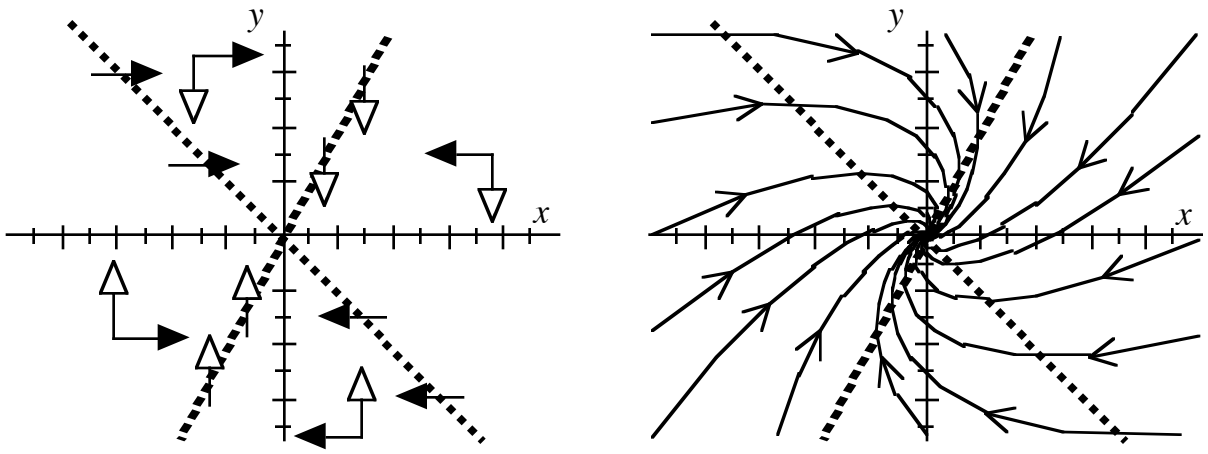


Figure 4.3. A linear 2-D system. Left: The nullclines (dashed) partition the plane into four regions. Each region is characterized by the signs of conjoined arrows $\dot{x}$ (filled arrowheads) and $\dot{y}$ (open arrowheads). Note that there is only one type of arrowhead on the nullclines (since the other derivative is zero). **Right:** Trajectories are shown for the same system (Equation 4-2) with the nullclines. Note the horizontal and vertical directions of the trajectories at the points where they cross the nullclines.

Computer Simulation of First Order Dynamical Systems

The above discussion, including state spaces, nullclines, stable and unstable equilibria, and attractor regions is just an introduction to some concepts from dynamical systems[16]. While formal study of dynamical systems is just a natural extension of freshman calculus and linear algebra, there may be many readers who are not inclined to pursue this study. If you are among the daunted, fear not! You have been saved by the magic of computer simulation! Any first order system can be simulated with just a simple iterative loop consisting of just a few lines of code!

Since, for small time increment Δt, the ratio $\Delta x_i/\Delta t$ is approximately equal to the derivative $\dot{x}$, the behavior (i.e. trajectory) of any dynamical system $\dot{x}_i = f_i(x_1,\ldots,x_n)$ beginning from an initial state $\mathbf{x}_0$, can be approximated by a difference equation of the form $\Delta x_i = \dot{x}_i \Delta t = f_i(x_1,\ldots,x_n)\Delta t$. Thus, the change in the state variable x_i from a given time t to a short time later, $t+\Delta t$, is close to the quantity $f_i(x_1,\ldots,x_n)\Delta t$, or in more mathematical terms:

$$x_i(t+\Delta t) = x(t) + f_i(x_1,\ldots,x_n)\Delta t \qquad [4\text{-}3]$$

This is easily translated to computer code as an iterative loop:

---

[16] The interested reader is encouraged to learn more about dynamical systems either by taking a semester course in a college math department or by self-study. An excellent introduction can be found in Stolniz (199x) or in the appendix of Segal (198x).

Example 4-2. Gravity. An object released from a height falls with approximately constant acceleration. Here, let the state of a system moving in 1 dimension be defined by two variables position (s) and velocity (v), where velocity is just the first time derivative of the position. Thus our first equation is simply that velocity is the time derivative of position. The second equation is that the acceleration (the derivative of the velocity) is a negative constant:

$$\dot{s} = v$$
$$\dot{v} = a \quad (a \text{ is a constant})$$

Thus the following computer code simulates the distance fallen by an object released at 0 velocity from a height of 100 feet for a total of 2 seconds.

```
delta_t := 0.2 ;
nsteps := 10 ;
acceleration := -32 ;
s := 100 ;
v := 0 ;
for time := 1 to nsteps do begin
   delta_s := v * delta_t ;
   delta_v := acceleration*delta_t;
   s := s + delta_s ;
   v := v + delta_v ;
   end ;
```

The following table shows how this simulation progresses for various combinations of **`delta_t`** and **`nsteps`**, all of which multiply to 2 seconds; the exact answer would be a fall of 64 feet from the initial 100, giving $s(t{=}2 \text{ sec}) = 36$ feet.

$\Delta t = 0.2$

t	dv	ds	v	s
0.0	0	0	0	100.00
0.2	-6.4	0	-6.4	100.00
0.4	-6.4	-1.28	-12.8	98.72
0.6	-6.4	-2.56	-19.2	96.16
0.8	-6.4	-3.84	-25.6	92.32
1.0	-6.4	-5.12	-32.0	87.20
1.2	-6.4	-6.40	-38.4	80.80
1.4	-6.4	-7.68	-44.8	73.12
1.6	-6.4	-8.96	-51.2	64.16
1.8	-6.4	-10.2	-57.6	53.92
2.0	-6.4	-11.5	-64.0	42.40

$\Delta t = 0.5$

t	dv	ds	v	s
0.0	0	0	0	100
0.5	-16	0	-16	100
1.0	-16	-8	-32	92
1.5	-16	-16	-48	76
2.0	-16	-24	-64	52

$\Delta t = 1.0$

t	dv	ds	v	s
0.0	0	0	0	100
1.0	-32	0	-32	100
2.0	-32	-32	-64	68

```
for i:= 1 to n do x[i] := x0[i];                    {initialize x}
for time := 1 to nsteps do begin
  for i := 1 to n do dx[i] := f(i,x)*dt ;           {compute dx/dt}
  for i := 1 to n do x[i] := x[i] + dx[i] ;             {update state}
  end;
```

Note that Δx_i has to be computed for all components x_i before any of them are updated (consideration of the alternative is left as an exercise at the end of the section). Of course, the accuracy of the approximations improves with *smaller* values of Δt, but since the amount of time simulated is the number of iterations through the loop times the value of Δt, a *larger* value of Δt will lead to a shorter simulation of a given total time period T. Thus, there is an intrinsic tradeoff between simulation speed and accuracy modulated by the program parameter **`delta_t`**. Consider the simple dynamical system describing free fall in a vacuum. [17]

Thus, the behavior of dynamical systems can be explored experimentally for systems that elude formal analysis, as is the case for most nonlinear systems. We shall see how learning in neural networks can be characterized in terms of dynamical systems, and that the application of computer simulation is hence straightforward in principle (and straightforward in practice as well, as long as we are careful about the details).

4.2. Gradient Optimization

Suppose we seek an extreme value (i.e., the minimum or the maximum) of a continuous unknown function $f(x)$. Short of evaluating the function for every value of the argument x, we must make certain assumptions about the form of the function. The essence of the gradient technique is to "guess" an initial value for x, and then change x by a small amount, such that the resulting change in $f(x)$ is in the desired direction. If a value x_m is found such that any infinitesimal change in x (positive or negative) results in an undesirable change in $f(x)$, then $f(x_m)$ is a *local extremum* ; that is, in its immediate neighborhood, it is the minimum (or maximum) value for the function x.

---

[17] A more common, and entirely equivalent, dynamical system for expressing motion under constant acceleration is in terms of the *second* time derivative of position: $\ddot{s} = a$. This is an instance of a very useful property of dynamical systems expressed in terms of higher order time derivatives; namely, any equation of the form $\ddot{x}_i = g_i(x_1,\ldots, x_n)$ can be reduced to a coupled pair of first order equations by intoducing a new variable $v_i \equiv \dot{x}_i$. Thus, the two coupled first order equations are:

$$\dot{v}_i = g_i(x_1,\ldots, x_n)$$

$$\dot{x}_i = v_i.$$

Underlying the assumption is that the function is smooth at the scale of the small changes made to x; that is, the function have at most one extreme value within one such step.

In direct analogy to the dot-notation for time derivatives (described in the preceding section), the derivative of $f(x)$ with respect to x is denoted $f'(x)$. Recall that the derivative is the rate of change of $f(x)$ with respect to x; if it is positive, $f(x)$ is increasing with increasing x, if it is negative, $f(x)$ is decreasing with increasing x. The case where $f'(x) = 0$ often indicates a local minimum or maximum, but the derivative can also be zero at "points of inflection", where $f'(x)$ goes from positive to zero, and is then positive again; or alternatively, $f'(x)$ goes from negative to zero, returning to negative.

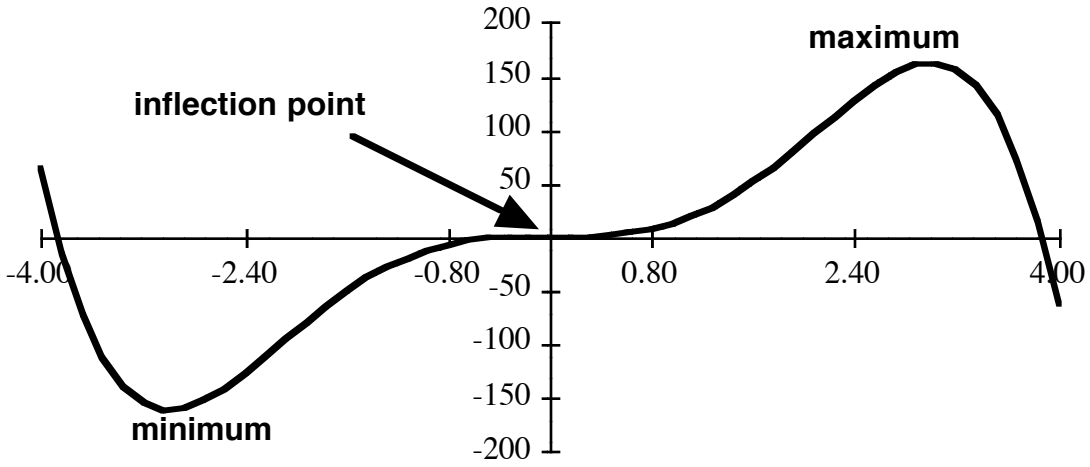


Figure 4.4. Zero derivatives. An example of a function with local minimum, a local maximum, and an inflection point, all of which have zero values for the derivative. The plotted function is: $f(x) = -x^5 + 15x^3$.

If a maximum to $f'(x)$ is sought, it is appropriate to change x in proportion to $f'(x)$; if the goal is to minimize $f(x)$, then it is appropriate to change x in proportion to $-f'(x)$. Thus, we can posit a dynamical system of the following form:

$$\dot{x} = \begin{cases} f'(x) & \text{for maximization} \\ -f'(x) & \text{for minimization} \end{cases} \qquad [4\text{-}4]$$

Thus, a value of x for which $f'(x)$ is zero is an *equilibrium point* for x, since $\dot{x}$ is zero at these points; this occurs whenever x is a minimum or maximum point. For a minimization process, the local minima are stable equilibria, and the local maxima are unstable, (and vice versa for a maximization process (see Figure 4.5)).

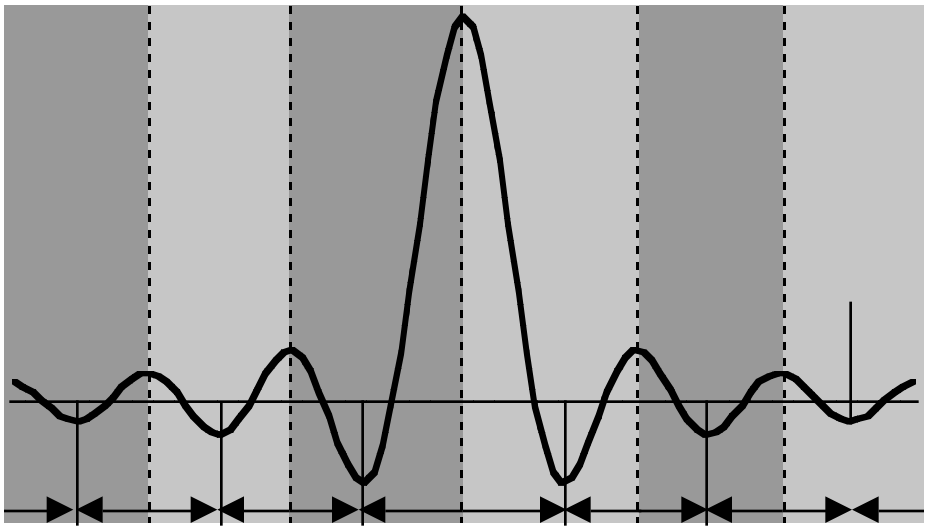


Figure 4.5. Local minima and maxima. This damped sine wave, $f(x) = \sin(x)/x$, has several local maxima and minima. Each local minimum m_i has a "capture domain" (shaded regions) $\mathbf{C}_i$. A gradient descent process with an initial point anywhere within $\mathbf{C}_i$ will converge to m_i.

Application of the chain rule from calculus shows the effect of gradient descent on the value of $f(x)$. The prescribed change in x for maximization of $f(x)$ results in either a positive change or no change to f, and the change in x for minimization of $f(x)$ either decreases f, or leaves it constant:

$$\Delta f(x) = \frac{df(x)}{dx}\Delta x = \begin{cases} -\eta\left(\frac{df(x)}{dx}\right)^2 \le 0 & \text{for minimization} \\ +\eta\left(\frac{df(x)}{dx}\right)^2 \ge 0 & \text{for maximization} \end{cases} \qquad [4\text{-}5]$$

The *step size* parameter η (also called the *learning rate*)regulates the speed and accuracy of the process; note that η corresponds to the time step parameter Δt (`delta_t` in the code from the last section). If η is too small, the process may not converge to a solution in an acceptable time; too large, and x may jump over a desirable extremum.

The following PASCAL program implements this procedure.

```
program optimize(input,output) ;
ctr, nstep : integer ;
eta, x, fval, slope : real ;

function func(x:real) : real ;
begin
   [define the function f(x) here]
end;

function fder(x:real) : real ;
begin
   [give an expression for the derivative f'(x) here]
```

```
end;

procedure report ; {this could be graphical instead of tabular}
begin
   fval := func(x) ;
   slope := fder(x) ;
   writeln(ctr:4, x:11:4, fval:11:4, slope:11:4) ;
end;

begin
   write('Input step size and number of steps:') ;
   readln(eta, nstep) ;
   write('Initial value for x: ') ;
   readln(x) ;
   report ;
   for ctr := 1 to nstep do begin
      x := x + eta*fder(x) ; {+ for maximum, - for minimum}
      report
      end ;
end.
```

Consider example 4.3.

Local extremes

Gradient descent techniques can be easily led into *local* optima; these are points that are the best in some local area, but not as good as some point more distant. Usually the *global* optimum or something nearly as good, is sought. The function $f(x) = (\sin x)/x$ has an infinite number of local minima (4.5, above). Note that gradient descent can converge to any of these, depending on the choice of x_0.

Local minima and maxima are a fundamental problem plaguing gradient descent techniques. They can be partailly avoided using some techniques, but no enhancement *guarantees* a global optimum.

In certain cases, an exact formula for the derivative cannot be found. Here, it should be noted that the derivative can be estimated by perturbing x by a small amount Δ; note that this expression becomes the derivative in the limit, as Δ approaches 0. In general, Δ should be much smaller than the step size η.

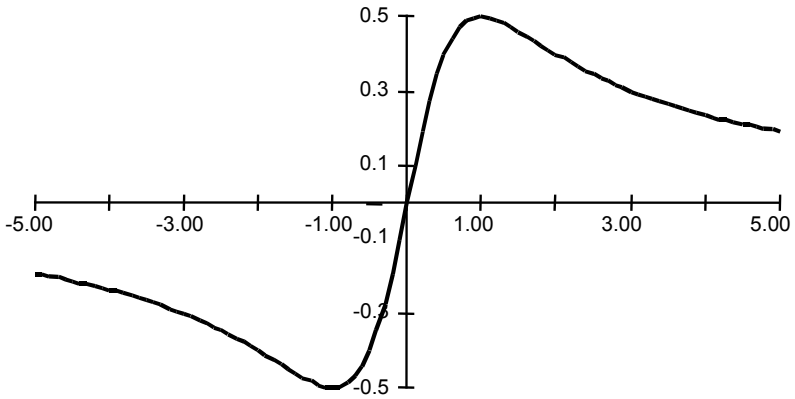
$$f'(x) \approx \frac{f(x+\Delta) - f(x)}{\Delta} \quad \text{where } \Delta << \eta \qquad [4\text{-}6]$$

This is implemented by the code below, which can be substituted into the program *optimize* above.

```
const eps = 0.0001 ;

function fder(x:real) : real ;
begin
   fder := ( func(x+eps) - func(x) ) / eps ;
end;
```

Example 4.3. The function $f(x) = \frac{x}{x^2 + 1}$ has a minimum value and a maximum value. Use gradient optimization to find $x_{\min}$ and $f(x_{\min})$.



Solution. Using elementary differential calculus, the derivative is obtained:

$$f'(x) = \frac{1 - x^2}{\left(x^2 + 1\right)^2}$$

The extrema of this function occur where the first derivative is zero, which is easily seen to be at $x = \pm 1$. First we seek the maximum. Let us make an initial guess of $x_0 = -2$ for seeking the minimum. Note the performance of the gradient descent technique for various values of η.

$\eta = 1$

iteration	x	f(x)	df/dx
0	-2.000	-0.400	-0.120
1	-1.880	-0.415	-0.123
2	-1.757	-0.430	-0.125
3	-1.632	-0.446	-0.124
10	-1.038	-0.500	-0.018
11	-1.020	-0.500	-0.010
12	-1.010	-0.500	-0.005
13	-1.005	-0.500	-0.003
14	-1.003	-0.500	-0.001

$\eta = 2$

iteration	x	f(x)	df/dx
0	-2.000	-0.400	-0.120
1	-1.760	-0.430	-0.125
2	-1.510	-0.460	-0.119
3	-1.272	-0.486	-

iteration	x	f(x)	df/dx
0	-2.000	-0.400	-0.120
1	-1.400	-0.473	-0.110
2	-0.852	-0.494	0.092
3	-1.312	-0.482	-0.097
4	-0.825	-0.491	0.113

Gradient descent in two dimensions and beyond

The gradient optimization technique can be applied to functions of several variables. Consider a function of two variables, $f(x,y)$. The goal in this case is to find a pair of values (x,y) that minimize (or maximize) the function f. The *gradient* ∇f is a vector in the x-y plane which points in the direction of greatest change in f (the "steepest" direction) with a magnitude proportional to the steepness in that direction. It is defined as the vectors whose x and y components are respectively the partial derivatives of f with respect to x and y.

$$\nabla f \equiv \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$$

The gradient of f is a vector that is multiplied by a small constant, η, and added to the vector (x,y) to give a new vector (x,y), for which the gradient can be recalculated, and so on. Thus, the basic gradient descent procedure described earlier is extended to functions of two variables. If we denote the partial derivatives of f with respect to x and y by f_x and f_y respectively, the following expressions for the update of x and y lead to local minima in f (the minus sign would be deleted for maximizing f).

$$\Delta x = -\eta f_x(x,y) \quad \text{and} \quad \Delta y = -\eta f_y(x,y) \qquad [4\text{-}7]$$

Again the chain rule can be applied to demonstrate that these changes will always decrease f or leave it constant.

$$\begin{aligned}\Delta f(x,y) &= f_x(x,y)\Delta x + f_y(x,y)\Delta y \qquad \text{[the chain rule]}\\ &= -\eta (f_x(x,y))^2 - \eta (f_y(x,y))^2 \le 0\end{aligned}$$

If both derivatives (f_x and f_y) are zero, the point could be a minimum, a maximum, or a point of inflection, as in the single variable case. Another possibility is a so-called saddle point, for which the derivative in one direction is a maximum, and the derivative in the orthogonal direction is a minimum (see Figure 4.6, left). Of course, as in the single variable case, gradient descent in two variables is susceptible to the local minimum problem. That is, it is possible to converge to a point (x,y) that yields a value for f that is smaller than the value for any other point in the immediate neighborhood, but is greater than that given by another relatively distant (x,y). Consider a function that looks like a smooth plane dotted with dimples of varying depth (see Figure 4.6, right).

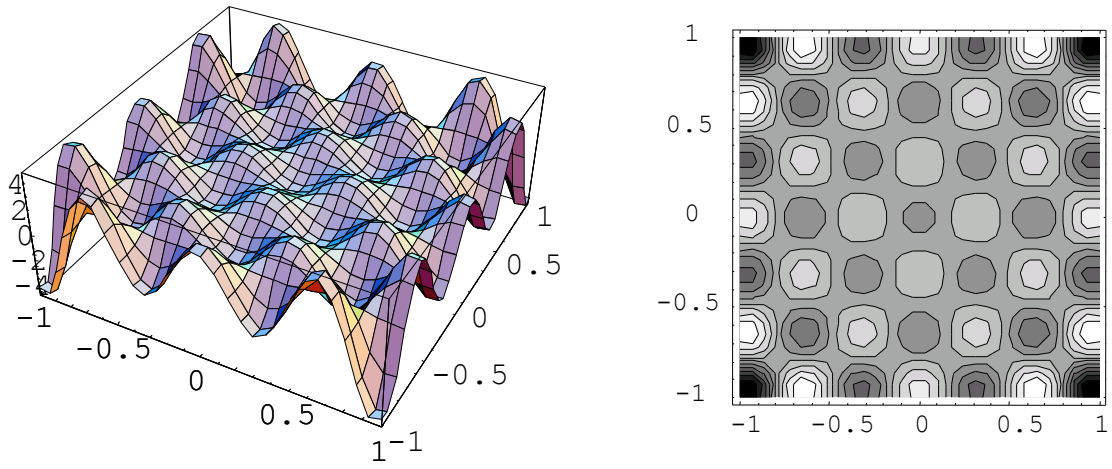


Figure 4.6. Gradient descent on functions of two variables. **Left**: The function $f(x, y) = 1 - \exp(-x^2 - y^2)\cos 5x \cos 5y$ is plotted. **Right**: A contour plot of the function (dashed contours) with trajectories (solid). Note that each minimum "captures" all trajectories in its immediate neighborhood.

The PASCAL program below implements gradient descent in two dimensions.

```
program opt2var(input,output) ;
ctr, nstep : integer ;
eta, x, fval, gradx, grady : real ;

function func(xx,yy:real) : real ;
begin
   [give an expression for the function here]
end;

function fderx(xx,yy:real) : real ;
begin
   [define the function for the derivative fx(x,y) here]
end;

function fdery(xx,yy:real) : real ;
begin
   [define the function for the derivative fy(x,y) here]
end;

procedure report ; {this could be graphical instead of tabular}
begin

fval := func(x,y) ;
gradx := fderx(x,y) ;
grady := fdery(x,y) ;
writeln(ctr:4, fval:11:3, gradx:11:4, grady:11:4) ;
end;

begin
   write('Input step size and number of steps:') ;
```

```
      readln(eta, nstep) ;
      write('Initial values for x and y: ') ;
      readln(x,y) ;
      report ;
      for ctr := 1 to nstep do begin
         dx = eta*fderx(x,y) ;
         dy := eta*fdery(x,y) ;
         x := x + dx ; {+ for maximum, - for minimum}
         y := y + dy ; {+ for maximum, - for minimum}
         report
         end ;
end.
```

As in the single variable case, if circumstances indicate a need, the derivatives can be approximated.

$$f_x(x,y) \approx \frac{f(x+\Delta,y) - f(x,y)}{\Delta}$$
$$f_y(x,y) \approx \frac{f(x,y+\Delta) - f(x,y)}{\Delta} \qquad [4\text{-}8]$$

The extension of gradient descent beyond two dimensions is straightforward. Generally, for functions of N variables $f(x_1, x_2, \ldots, x_N)$, gradient descent is performed with respect to each variable, resulting in a dynamical system with N equations:

$$\dot{x}_i = -\frac{\partial f(x_1, x_2, \ldots, x_N)}{\partial x_i} \quad \text{for } i = 1, \ldots, N \qquad [4\text{-}9]$$

4.3. Minimizing Summed Squared Error

In Section 1.3, the principle of finding parameters that minimize error was presented, and demonstrated for certain types of functions where the equation $\partial E / \partial p_i = 0$ could be solved directly for the parameters that minimize E. For more general types of functions, there are no techniques for exactly solving this equation. However, we can use gradient descent to find minimal values of E; of course, we can not assume that these are *global minima* (but we can hope!). In many cases, even if the parameters are not globally optimal, they are good enough for the purpose at hand (whatever that may be), and they are often better than what alternative techniques might yield.

We can perform gradient descent in the error in order to find a set of parameters that optimize a fitness function. Recall the definition of the summed squared error, where the function to be optimized is now denoted $f(x_i;p_1,...,p_K)$, and f is a function of x_i with parameters p_k, $k=1...K$. The data points are pairs of the form (x_i,y_i) and the p_k are to be optimized to fit the data.

$$E = \sum_{i=1}^{N} (y_i - f(x_i; p_1 \ldots p_K))^2 \qquad [4\text{-}10]$$

Applying the principle of gradient descent, a dynamical system can be derived[18]:

$$\dot{p}_k = -\frac{\partial E}{\partial p_k} \qquad [4\text{-}11]$$

Hence the prescribed change in p_k is obtained by computing a difference between y_i and $f(x_i;p_1,...,p_K)$ for each data point (x_i, y_i), and summing the differences over i. It is often more practical (due to large data sets, or limitations in computer speed or memory), to select implement parameter changes based on differences computed from *single data items*. By summing the differences first (the *batch* mode of learning), trajectories in parameter space are generally smoother, but may result in slower convergence than changing the parameters from individual data items (*online* mode), which are usually selected at random.

$$\Delta p_k = \eta \sum_i (y_i - f(x_i; p_1 \ldots p_K)) \frac{\partial f}{\partial p_k} \quad ; \quad k = 1 \ldots K \qquad [4\text{-}12]$$

There are a number of considerations inherent in iterative schemes for minimizing error, such as choosing an appropriate value for the step size parameter η and choosing a stopping criterion. The stopping criterion is typically expressed as an "**OR**" condition: The programs stops if either the error value reaches an acceptable level (usually near zero), *OR* if a maximum number of iterations is reached (in order to guarantee that the program ends). There is no formula for the optimal choices of these program parameters.

If the number of data items (N) is very large (in some cases data arrives continuously, so N is infinite), gradient descent can be individually applied to the error for each item. Let us denote the change in parameter for item i, by $\Delta^i p_k$. Also, let the error generated by item i, be denoted E^i. That is,

$$E = \sum_{i=1}^{N} E^i \qquad [4\text{-}13]$$

Thus, the general recipe for error minimization by gradient descent is as follows (by the online technique):

$$\Delta^i p_k = -\eta \frac{\partial E^i}{\partial p_k} = 2\eta (y_i - f(x_i; p_1 \ldots p_K)) \frac{\partial f(x_i; p_1 \ldots p_K)}{\partial p_k} \quad \text{for } i = 1 \ldots N \qquad [4\text{-}14]$$

---

[18] A function which decreases over all trajectories of a dynamical system, and is bounded from below, is called a *Lyapunov Function* of that system. If such a function exists for a system, that system is guaranteed to reach (or asymptotically approach) a stable equilibrium point. All gradient descent systems have as a Lyapunov function the function that is being minimized.

Numerical solution of the system can be found by approximating the system as follows (note that since the constant η is arbitrarily chosen, the factor of 2 can be eliminated). Thus, we apply gradient descent as described earlier, by first initializing the parameters to random values, then computing changes in each parameter according to this recipe:

1. Determine a suitable parametric function.
2. Select step size (η), stopping criterion, and initial parameter values.
3. Determine changes in the parameter values using gradient descent (either batch or online).
4. Repeat step 3 until the stopping criterion is reached.

Example 4.3: Use gradient descent to find a circle that fits a given set of data.

Recall (from high school math) that the (x,y) points on a circle with a center at (c_x,c_y) and radius r can be described by the expression $(x-c_x)^2 + (y-c_y)^2 = r^2$. Thus, there are three parameters that describe a circle: (c_x,c_y,r). We begin with a summed square error function:

$$E = \sum_{i=1}^{N} \left(r^2 - (x_i - c_x)^2 - (y_i - c_y)^2 \right)^2 = \sum_{i=1}^{N} Q_i^2 \quad \text{where} \quad Q_i \equiv r^2 - (x_i - c_x)^2 - (y_i - c_y)^2$$

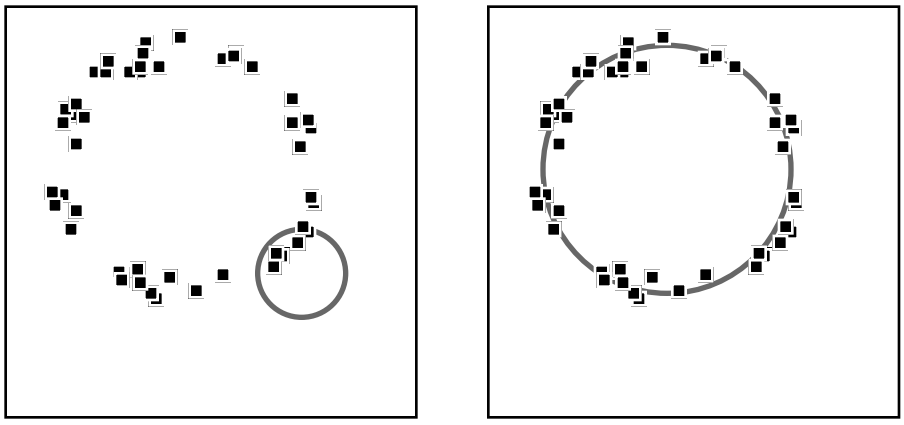
and compute derivatives with respect to each parameter:

$$\frac{\partial E^i}{\partial r} = 4rQ_i \qquad \frac{\partial E^i}{\partial c_x} = 4(x_i - c_x)Q_i \qquad \frac{\partial E^i}{\partial c_y} = 4(x_i - c_y)Q_i$$

Thus, a circle can be fit to a set of data by starting with arbitrary values for r, c_x, and c_y, and iteratively updating them according to the equations

$$\Delta r = \eta r Q_i \qquad \Delta c_x = \eta (x_i - c_x) Q_i \qquad \Delta c_y = \eta (x_i - c_y) Q_i$$

The figure shows a roughly circular data set with the circles (gray) determined by the initial (left) and final (right) sets of the three parameters:



4.4. The Cost Function

So far, we have seen how the summed squared error E can be minimized by computing gradients of E, and finding states at which they are zero. Alternatively, other functions of the weight parameters may be defined, and optimized; these are generally called *cost functions*. The cost function need not be the summed squared error. In this section, some other measures that also optimize the match between the actual and desired function values are described. One class of cost functions, of which the SSE is a special case, is defined by summing the differences between the data values y_i and the approximations $f(x_i; p_1 \ldots p_K)$ raised to any power m, where m is an *even integer*.

$$E = \sum_{i=1}^{N} (y_i - f(x_i; p_1 \ldots p_K))^m \qquad [4\text{-}15]$$

The derivatives with respect to each parameter thus have the form

$$\Delta p_k = -\eta \sum_i (y_i - f(x_i; p_1 \ldots p_K))^{m-1} \frac{\partial f}{\partial p_k} \quad ; \quad k = 1 \ldots K \qquad [4\text{-}16]$$

where a factor m has been absorbed into the learning rate. In general, it is best to keep m as small as possible (that is, m=2), since raising a small difference between y_i and $f(x_i; p_1 \ldots p_K)$ to a high exponent results in a *very* small value for the change, thus slowing convergence (see Figure 4.7).

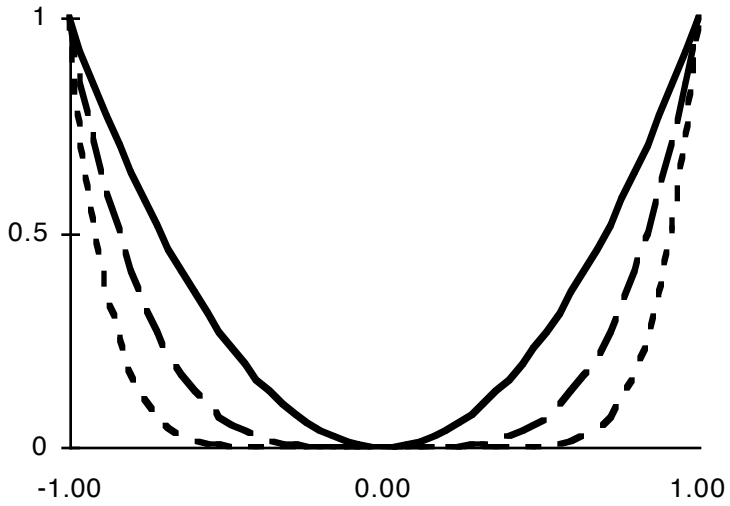


Figure 4.7. Power functions. The plots show $(y\text{-}f)^m$ as a function of $(y\text{-}f)$, for m =2 (solid), m =4 (long dashes), and m=8 (short dashes). Note that the minimum is always at zero, but that as m increases, the

functions become flatter. Since convergence speed is proportional to the slope, higher exponents result in slower convergence.

Of course, not all cost functions are power functions. The *Kuhlbach-Lieber* measure[19] is an attractive option for classification tasks in which the data values y_i take on the values 0 and 1, and the function f takes on values in the range [0,1]; the value of f can then be interpreted as the probability that the input is in the class "$y = 1$".

$$E^{KL} \equiv -\sum_{i=1}^{N} \left\{ y_i \ln f(x_i; p_1 \ldots p_K) + (1 - y_i) \ln(1 - f(x_i; p_1 \ldots p_K)) \right\} \qquad [4\text{-}17]$$

Like the SSE, E^{KL} is never less than zero, and is only zero when $f(x_i ; p_1 ... p_K)$ equals y_i for all i.

Terms can be added to the cost function, in cases where error minimization is not the sole concern; for example, the optimization of other quantities can be included in the cost function, simply by adding them as follows:

$$C = k_1 E_1 + k_2 E_2 + \ldots + k_n E_n = \sum_{j=1}^{n} k_j E_j \qquad [4\text{-}18]$$

where the E_j are various component quantities that must all be minimized and the k_j are coefficients that modulate the relative importance of the components.

Three items to note are:

[1] All the E should have lower bounds, otherwise, C can be driven to an arbitrarily low value by reducing a single (unbounded) E component sufficiently low without regard to the other components.

[2] Since the derivative is a linear operator, the derivative of a cost function formed as a weighted sum of component functions is equal to the weighted sum of derivatives of the component functions using the same weight coefficients. That is, the gradient descent rule $\Delta p_i = -\eta \partial C / \partial p_i$ can be written

$$\Delta p_i = -\eta_1 \frac{\partial E_1}{\partial p_i} - \eta_2 \frac{\partial E_2}{\partial p_i} - \ldots - \eta_n \frac{\partial E_n}{\partial p_i} = -\sum_{j=1}^{n} \eta_j \frac{\partial E_j}{\partial p_i}$$

---

[19] The *Kuhlbach-Lieber* measure is based on the *cross entropy* measure $CE(\mathbf{p},\mathbf{q})$, which gives a "distance" between probability distributions $\mathbf{p}$ and $\mathbf{q}$. If $\mathbf{p}$ and $\mathbf{q}$ are identical distributions, CE simply becomes the entropy of the distribution $H(\mathbf{p})$.

$$CE(\mathbf{p},\mathbf{q}) \equiv -\sum_{i} p_i \log q_i \quad \geq \quad H(\mathbf{p}) \equiv -\sum_{i} p_i \log p_i$$

where each η_i is the product of the "master" learning rate η and the coefficient k_i.

[3] It is possible to transform any constraint among the parameters into a component of the cost function. The latter observation is easily demonstrated by the following example, which is somewhat general. Suppose the parameter set $\{p_1,p_2,...\}$ is constrained by an equation of the form $f(p_1,p_2,...) = g(p_1,p_2,...)$. This constraint can be incorporated into the cost function by including a component of the form

$$E_j \equiv (f(p_1 \cdots p_N) - g(p_1 \cdots p_N))^2$$

where a corresponding constant k_j is chosen to regulate the importance of the constraint relative to the cost function. Thus, the constraint is not a "hard" constraint, in that the parameters need not be in strict compliance; instead, this *soft constraint* is an influence on the weights that can be modulated relative to the other terms in the cost function using the coefficient k_j.

Example 4.4. Derive a procedure that minimizes the error of a linear unit under the soft constraint that the sum of the squares of the parameters is 1.

An error term E^{norm} can be defined by $E^{norm} \equiv \left(1 - \sum_i w_i^2\right)^2$.

The corresponding term in the modification rule is given by $-\eta_{norm}\frac{\partial E^{norm}}{\partial w_i} = 4\eta_{norm} w_i \left(1 - \sum_j w_j^2\right)$.

The factor of 4 can be absorbed into the learning rate for the second term to give:

$$\Delta w_i = \eta_1\left(T^\eta - r^\eta\right)s_i + \eta_2 w_i\left(1 - \sum_j w_j^2\right)$$

4.5. Other Optimization Techniques

While backprop is based on gradient descent, the reader is advised that gradient descent is not the only method for optimizing functions; in fact, in many situations it isn't even a particularly good one. Two other techniques that have been applied to minimizing error in neural networks are *Newton's method* and the *conjugate gradient* technique (Hestenes and Stiefel, 1952). For the sake of comparison and completeness, these methods are briefly described in this section.

Newton's Method

For many functions, Newton's Method (NM) converges much more quickly than gradient descent; on the downside, NM is less likely to converge smoothly and will not infrequently converge to a local maximum rather than a global minimum!. Newton's method is actually designed to find values of the independent variable(s) for which the function has a value of zero, so we apply it to the derivative to find states where $f'(x) = 0$, which is not guaranteed to be a minimum, even locally.

Let x^* be a zero of the function $f(x)$; that is, x^* satisfies the condition $f(x^*)=0$. Like gradient descent, NM begins with an initial guess x_0 and approximates x^* iteratively using successive approximations x_1, x_2, x_3... based on computation of the derivative. Given x_0, and the corresponding values $f(x_0)$ and $f'(x_0)$, an equation can be determined for a line that passes through the point $(x_0,f(x_0))$ with slope $f'(x_0)$, so it is tangent to the curve $f(x)$. Using the equation $y-b=m(x-a)$ for a line of slope m that passes through the point (a,b), we get the following formula for x_{i+1} in terms of x_i:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \qquad [4\text{-}19]$$

A very naive approach would be to use this formula to directly seek a zero in the error, but for many tasks, the error function is never zero. In such cases, NM can diverge (see Figure 4.8), so we are generally much better off seeking zeroes in $f'(x)$. Thus, we apply NM to the derivative, which simply means

$$x_{i+1} = x_i - \frac{f'(x_i)}{f''(x_i)} \quad \text{or} \quad \Delta x = -\frac{f'(x_i)}{f''(x_i)} \qquad [4\text{-}20]$$

Note that this iterative formula has the potential for unpredictable or erratic behavior at states where the second derivative is close to zero, for example near a maximum or a local minimum that is nonzero.

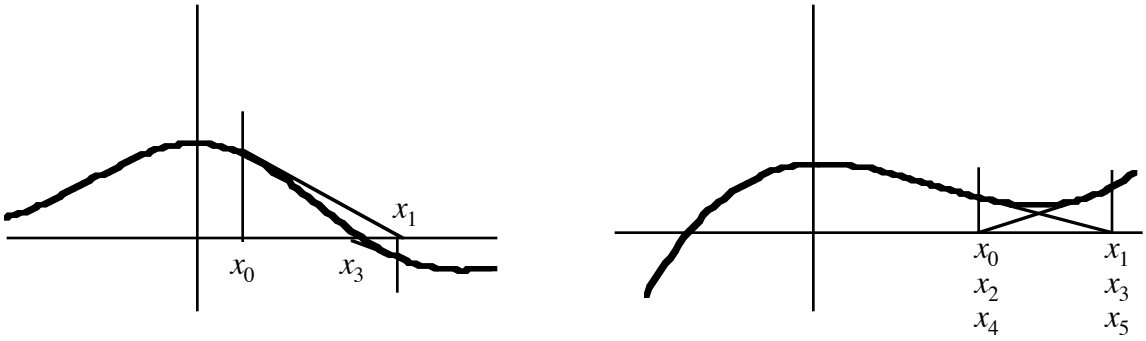


Figure 4.8 Convergence of Newton's Method. Convergence of NM to a zero can be rapid and smooth (left), but near a point where the derivative is zero or very small, as it is near a local minimum, the NM procedure can be non-convergent (the right plot oscillates between two values).

Whereas Newton's method applied to a function of one variable requires computation of one first derivative and one second derivative at each time step, optimizing functions of N variables (with N>1) requires computation of N first derivatives and N^2 second derivatives. The Hessian matrix $\mathbf{H}$ is defined such that its elements H_{ij} are the second derivatives of E with respect to parameters x_i and x_j.

$$\mathbf{H} \equiv \begin{bmatrix} \frac{\partial E}{\partial x_1 \partial x_1} & \frac{\partial E}{\partial x_1 \partial x_2} & \cdots & \frac{\partial E}{\partial x_1 \partial x_N} \\ \frac{\partial E}{\partial x_2 \partial x_1} & \frac{\partial E}{\partial x_2 \partial x_2} & \cdots & \frac{\partial E}{\partial x_2 \partial x_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial x_N \partial x_1} & \frac{\partial E}{\partial x_N \partial x_2} & \cdots & \frac{\partial E}{\partial x_N \partial x_N} \end{bmatrix} \qquad [4\text{-}18]$$

By analogy to the single variable formula [4-17], the formula for Newton's method in N dimensions can be expressed in terms of the vector $\mathbf{x}$:

$$\Delta\mathbf{x} = \mathbf{H}^{-1}(\mathbf{x})\nabla E(\mathbf{x}) \qquad [4\text{-}19]$$

Thus the quotient of the first derivative and the second derivative in [4-17] is seen as a special case of the multiplication of the gradient vector $\nabla E(\mathbf{x})$ by the inverse of the Hessian matrix. Since computation of the inverse requires on the order of N^3 multiplications at each iteration, the NM procedure can be unwieldy for large N. If the off-diagonal elements of $\mathbf{H}$ are sufficiently small, they can be ignored and we are left with an approximation that resembles the NM rule for a function of one variable:

$$\Delta x_i \approx \frac{\frac{\partial E}{\partial x_i}}{\frac{\partial^2 E}{\partial x_i^2}} \qquad [4\text{-}20]$$

Care should be taken in the application of this rule to guard against the possibility that the denominator is not too small, since that could produce too large a change in x_i. Also note that a negative value for the denominator will change x_i in a direction opposite to the downhill gradient.

Line Search and Conjugate Gradient Methods

The change in the state variable $\mathbf{x}$ at each iteration is determined by the gradient times the step size parameter. Let us define the vector $\mathbf{g}$ to be the direction of change in gradient descent (i.e., opposite to the gradient direction ∇E). In a *line search*, we seek the optimal amount ("distance") to move in this direction. [The distance has been $\|\nabla E\|$ up to now.] The line in state space defined by the vector in the direction $\mathbf{g}$ that passes through the current point $\mathbf{x}^*$ includes all points $\mathbf{x}$ that satisfy the equation $\mathbf{x} = \mathbf{x}^* + k\mathbf{g}$ as k goes from $-\infty$ to ∞. Thus, line search boils down to

finding the value of k that minimizes the error along that line (see Figure 4.9). An optimal value of k is typically found by stepping through many values, computed E along the way, and stopping when E begins to increase; thus the value of k is only locally optimal.

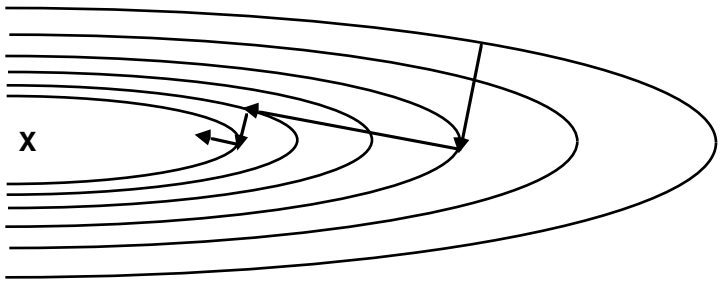


Figure 4.9. Line search. State changes (arrows) are shown against contours of constant error (ellipses). Each line search finds the state $\mathbf{x}(t+1)$ that minimizes the error function in the direction indicated by the gradient at a given state $\mathbf{x}(t)$.

As long as it is not zero, the gradient of any surface is orthogonal to the contour of constant error. Note (in Figure 4.26) that the line search converges to a point at which the gradient is tangent to such a contour. Thus, the subsequent gradient direction is orthogonal to the preceding gradient direction; that is $\mathbf{g}(t)\cdot\mathbf{g}(t+1)=0$, resulting in a "zigzag" trajectory. *Conjugate gradient* methods attempt to smooth the path by including a component of the previous step in the new step. Here, we select $\mathbf{g}(t)$ by adding a fraction γ of the previous line direction to the gradient direction [4-21], then perform a line search in this direction.

$$\mathbf{g}(t) = -\nabla E(x(t)) + \gamma \mathbf{g}(t-1) \qquad [4\text{-}21]$$

The best value of γ is determined by satisfying the condition $\mathbf{g}(t)\cdot\mathbf{H}\mathbf{g}(t+1)=0$; if this condition is met, the two vectors $\mathbf{g}(t)$ and $\mathbf{g}(t+1)$ are called *conjugate*. The Polak-Ribiere rule [4-22] gives the optimal value of γ in terms of $\mathbf{g}(t)$ and $\mathbf{g}(t+1)$ without requiring computation of $\mathbf{H}$.

$$\gamma = \frac{\left(-\nabla E(x(t)) - -\nabla E(x(t-1))\right)\cdot -\nabla E(x(t))}{\left(-\nabla E(x(t-1))\right)^2} \qquad [4\text{-}22]$$

4.6. Exercises

1. The **second order** differential equation $\ddot{s} = -s$ describes a simple oscillator. Convert this to a pair of first order equations. Identify the nullclines and the equilibrium point of the system. Write a program to numerically simulate the system, and run the program using Δt values of 0.01, 0.1, and 1, with initial values for s and $\dot{s}$ of -1 and 0 respectively.

2. Use gradient optimization to find a maximum value for the function $f(x) = xe^{-x}$.

3. Write a program to perform gradient descent on a function of fifty variables, defined as an array $x[1]$ ...$x[50]$.

4. Use the gradient optimization program to find mimima for the damped sine wave. Try several initial points and demonstrate that each local minimum x_i^* lies in the midst of a "capture region", R_i, defined as a set of points that, if chosen as initial points, will generate trajectories that converge at x_i^*. Also, demonstrate that if the initial point is chosen near a region boundary, the convergence time is longer.

5. Write a program to minimize the function $f(x,y) = \sin(x)\cos(y)$, using gradient descent. Try using several initial starting points, with x and y in the range [-10,10]. Note that the capture region (defined in exercise 3) corresponding to each local minimum is a 2-D region in the x-y plane. Plot (by hand or computer) the local minima in the region and sketch the capture regions.

6. Using gradient descent, find parameters a and b that fit the function $f(x;a,b) = a\sin(bx)$ to the following (x,y) data: (0,0), (1,1), (2,2), (3,3). Plot the solution along with the four data points.

5. *Learning*

There are many ways of acquiring knowledge: learning by being told, learning by imitation, learning from mistakes. In each case, a system that has *learned* is different than it was before, and presumably it is better off. Learning is thus a *process* that should drive a system to behavior that is more *appropriate*; here this will mean an improvement in performance that is measurable. In order to learn, the system requires some sort of guidance; that is, there must be evaluative feedback from a source outside the system that provides the system with information it can use to alter its behavior, assuming there is room for improvement.

It is important to distinguish between *learning* and *table lookup* (see Figure 5.1). A "table lookup" system explicitly stores correct stimulus-response pairs, and matches incoming stimuli to find the appropriate response. Such a system can behave perfectly (i.e. always give the correct response in a particular situation) if the input can be found in the table; for example, if the space of possible patterns is sufficiently small, such that the correct response to the entire space of inputs can be explicitly stored. New knowledge can be acquired by adding **s-r** pairs to the table. Such a system has not *learned* in the sense that it cannot generalize its knowledge beyond the items stored in the table. If a system can generalize its knowledge so that it can find systematic relationships between the input and output variables, then it can be said to *learn.*

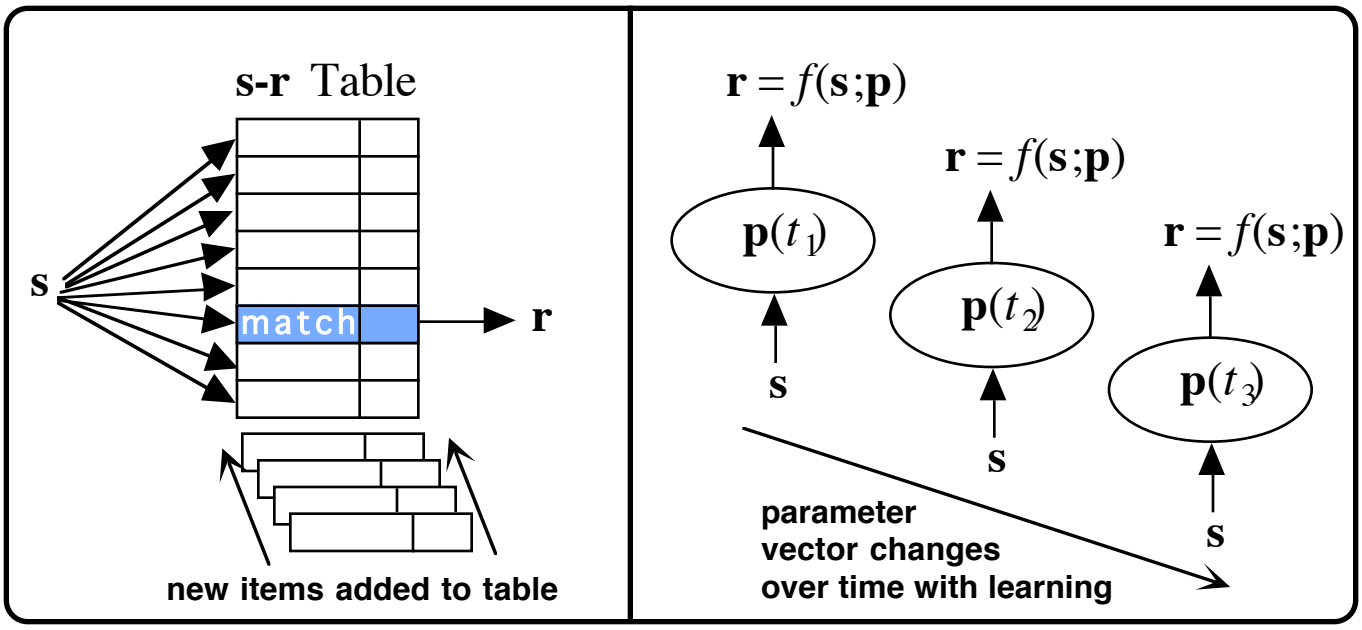


Figure 5.1. Table lookup vs. learning a function. A table lookup system (left) determines the response by comparing the stimulus **s** to table entries, and finding a match (shaded). An example of a learning system is a parametric function (right), whose parameter set **p** learns to produce a function that gives an appropriate response to **s**.

The idea of table lookup can be broadened to a much more useful technique known as nearest neighbor learning, in which the distance of a novel stimulus is computed to each of the stored items, with the closest item in the table treated as a "match" (described in Section 5.2). Learning a function of fixed capacity (fixed number of parameters) still has two distinct advantages over the nearest neighbor approach however, because as the number of items in the table becomes large:

- The computation of distance for the entire table of stored items can require too much *time*.
- The storage required for the table data can require too much *space*.

Learning in neural networks has generally been expressed in terms of dynamical systems, in which the state variables are the weights in the network; thus, learning is expressed as equations that specify changes in the weights.

5.1 Rules and Mappings

Let a **task** be defined as a set of stimulus-response pairs, where each stimulus and response consist of one or more real values, and the number of pairs that define the task can be finite or infinite. In solving a task, or designing a

system to solve it, the allowed values of the stimulus and response components are a very important aspect of the task. A feedforward neural network with N input components and M output units computes a **mapping** from an N dimensional space to an M dimensional space. Note that the input components are not limited in range by the processing function of the network, even though most tasks are conceived with a range in mind, whether or not it is explicitly stated; thus the stimulus space is generally ^{N}. The output space is determined by the range of the output units. Thus, linear output units have an infinite range ( ), linear threshold units have a binary range {0,1}, and sigmoids respond in the continuous range [0,1].

By exposing the network to task items, we hope to find a network mapping that closely matches the task. If the task can be easily described in terms of equations or "rules" that *systematically* relate the components of the output vector to those of the input vector, then it may be possible to express this relationship in the form of a neural network mapping. Capturing the systematic relationship between stimulus and response gives the network the potential for generalization. The relationship does not have to be 100% "clean"; that is some examples may violate the relationship. Thus, a task can be conceptualized as consisting of two parts -- a systematic component that is subject to generalization and a "noise" component that is inherently unpredictable. The systematic and noisy components are analogous to **rules** and **exceptions** in linguistic theories. Note that it can be difficult to know when one is dealing with a "true exception" or an example of a new "subrule". For example, students of English learning the rule "*e* follows *i*", may initially assume that "*receive*" is an exception, until they learn that there is a subrule (alternatively a class of "exceptions") stating that "*i* follows *e*" when the pair is preceded by *c*.

5.2 Learning, Testing, and Generalization

Achieving minimum error with respect to a set of data solves only half of the task of learning (at most!). In order to assess whether a system has "learned" about the relationships among the variables in a set of data, the system's performance must be tested on a set of data from the same distribution *that was not used to determine the parameters*. Thus, following the discussion in Chapter 3, we distinguish between two sets of data: a **training set**, $\mathcal{D}^{Tr}$, from which the system parameters are acquired (by regression or gradient descent or whatever), and a **test set**, $\mathcal{D}^{Ts}$,that is used to check the system's performance. Thus, since the cost function (for example, summed squared error) depends on both the choice of parameters and the data set, error values can be computed with respect to both data sets: E^{Tr} and E^{Ts}, respectively.

As a gradient descent process reduces the error with respect to the training set, the error will also be reduced in the test set to the extent that the system is learning the underlying relationships between the variables, which are expected to be reflected in both data sets. Another component of the error reduction reflects the system learning aspects of the data independent of the relationship between the variables; i.e. overfitting (see Chapter 3). Tracking

the errors on the two data sets during gradient descent often reveals two stages to learning: in the first stage, both E^{Tr} and E^{Ts} decrease, and in the second stage E^{Ts} begins to increase, while E^{Tr} continues to decrease (see Figure 5.2).

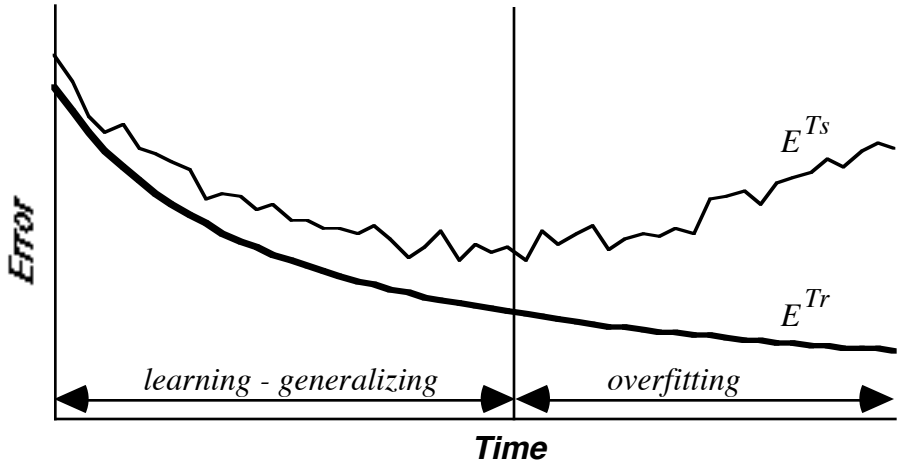


Figure 5.2. Training and Testing Errors. A typical scenario in which the error on the training set (bold curve) decreases over time, whereas the error on the test set (light curve) decreases for some period but ultimately increases as the learned parameter set overfits to the details of the training data. Due to the erratic nature of the test error, it can be very difficult to determine the precise point at which E^{Ts} begins to increase (the dashed line is approximate).

Thus, as the weight state converges to a minimal point on the training error surface, it does not converge to a minimum on the test error surface. The two error surfaces are sketched in Figure 5.3 with a trajectory that follows the gradient with respect to the training error.

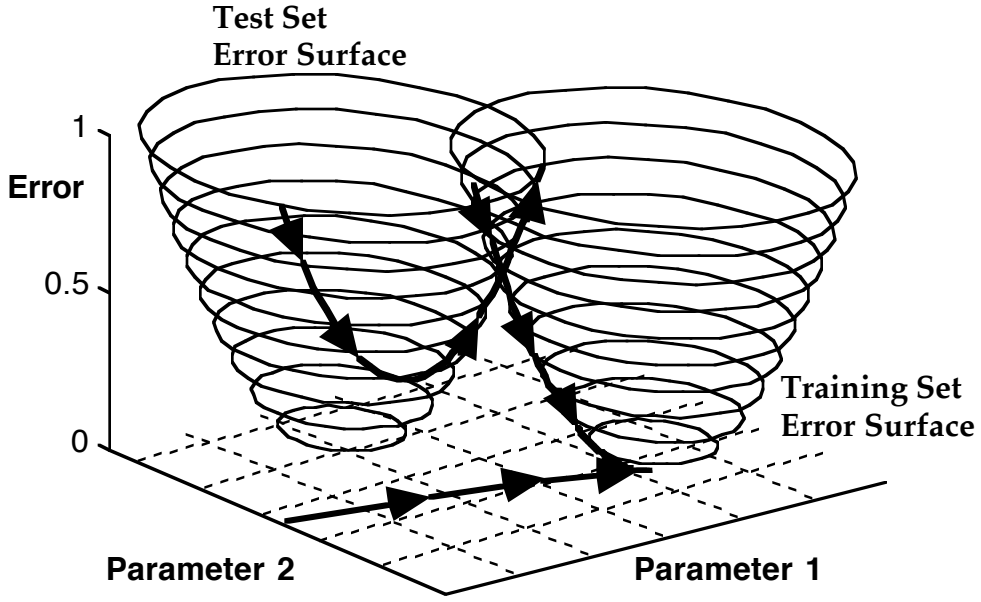


Figure 5.3. Training and Testing Error Surfaces. The trajectory in the parameter plane (bottom) traces error values along both the training (right) and testing (left) error surfaces. As it follows the gradient of the training surface, the test error initially decreases as the trajectory approaches both minima, then increases as the trajectory converges on the minimum of the training error surface.

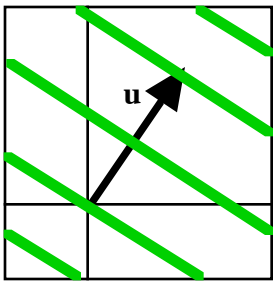
5.3 Nearest Neighbor

For purposes of contrast, a learning procedure is presented here that predates backpropagation, and in some ways complements it. The **nearest neighbor** technique has certain strengths where backprop has weaknesses, and vice versa. The idea is quite simple. All data items, both the stimulus variables and the corresponding target responses, are stored in memory exactly as they are encountered. Let a new stimulus to be evaluated be called a "probe"

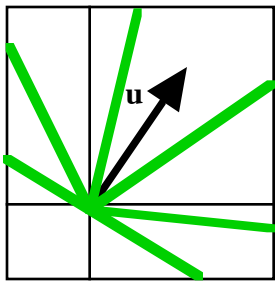
Distance and similarity

In order to find the nearest vector among a set to a given vector, a measure of distance or similarity must be defined. The concepts of distance and similarity are related: as the distance between two vectors increases, we tend to think of them as being less similar. Note that distance measures generally have the property that they are always greater than 0, except for the case of the distance between a point and itself, which is equal to 0. Thus, given any distance measure $\mathcal{D}(\mathbf{u},\mathbf{v})$, a similarity measure $\mathcal{S}$ can be defined as a decreasing function of $\mathcal{D}(\mathbf{u},\mathbf{v})$; for example $\mathcal{S} = exp(-\mathcal{D})$.

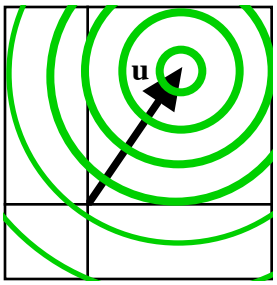
Consider the following few examples of similarity measures (note that S^{gauss} is an example of a similarity measure defined as a decreasing function of a distance measure. The figures show contours of constant similarity from a vector **u**:



$$S^{dot}(\mathbf{u},\mathbf{v}) \equiv \mathbf{u}\cdot\mathbf{v} \equiv \sum_i u_i v_i$$



$$S^{cos}(\mathbf{u},\mathbf{v}) \equiv \frac{\mathbf{u}\cdot\mathbf{v}}{|\mathbf{u}||\mathbf{v}|}$$



$$S^{gauss}(\mathbf{u},\mathbf{v}) \equiv e^{-(\mathbf{u}-\mathbf{v})\cdot(\mathbf{u}-\mathbf{v})}$$

stimulus. It is compared to each stored stimulus, and the distance[20] or similarity is computed to each. The decision of which neighbor is nearest, rests on how "nearness" is computed (see Box). Once a function for distance or similarity is chosen, the computed response is simply the target response that was given for the stored item that is closest to the probe.

Two PASCAL routines for the nearest neighbor procedure is given below: **assimilate** for storing new patterns, and **classify** for retrieving classifying novel patterns. The stored patterns are represented as rows of the matrix **s[ip,ic]**, and the corresponding classifications are in an array **class[ip]**. The selected distance function **dist(u,iv)** returns the distance between a vector **u** and row **iv** of **s**.

```
program nearest(input,output) ;
p, c : integer ;                                {indicate pattern and component}
npats : integer ;                               {number of stored patterns}
s : array[1..MAXPAT,1..ndim] of real ;          {stored patterns}
class: array[1..MAXPAT] of integer ; {classifications of patterns}

function dist(u: array[1..ndim] of real; iv:integer) : real ;
begin
   [give an expression for the function here]
end;

procedure assimilate(newone: array[1..ndim] of real; newclass: integer) ;
ii: integer;
begin
  if (npats<maxpat) then begin
   npats := npats+1 ;
   for ii:= 1 to ndim do s[npats,ii] := newone[ii] ;
   class[npats] := newclass ;
   end;
  else print("Pattern matrix FULL -- no more room!");
end;

function classify(newone: array[1..ndim] of real) : integer ;
begin
  for ii := 1 to npats do begin
   disti := dist(newone,ii) ;
   if (disti<mindist) then begin
    mindist := disti ;
    closepat := ii ;
    end ;
   end ;
  classify := class[closepat] ;
end;
```

[20] The distance between two vectors can be mathematically defined in many ways. Here, we use the Euclidean measure, with the distance between **u** and **v** defined as $d(\mathbf{u},\mathbf{v}) \equiv \sqrt{\sum_i (u_i - v_i)^2}$.

Nearest neighbor learning breaks the stimulus space into "tiled" regions $\mathcal{R}_i$, each of which is defined as the set of stimuli closer to a particular stored item, $\mathbf{s}^i$, than any other; thus any stimulus in $\mathcal{R}_i$ will be classified the same as $\mathbf{s}^i$. Figure 5.4 illustrates these regions for the data in Example 5.1, using a Gaussian distance measure. The boundaries between such regions are called *Voronoï* partitions. Interestingly, even though the contours of constant distance are circular (spherical for higher dimensional data), the partitions are piecewise linear.

Example 5.1. Given the stored points in the table below, classify the test points **X=**(1,4) and **Y**=(-2,0). Here, Euclidean distances are calculated, and the closest points are determined (shaded cells). Thus, (1,4) is assigned category **0**, and (-2,0) is assigned category **1**.

Stored Points				*Computed Distances to Test Points*	
i	$\mathbf{s}^i$		T^i	distance to **X**	distance to **Y**
1	-19	-3	0	21.2	21.2
2	-9	15	0	14.9	22.6
3	-11	-5	0	15.0	15.3
4	-7	3	1	8.1	9.5
5	3	7	0	3.6	9.7
6	3	-7	1	11.2	12.2
7	7	15	1	12.5	21.1
8	17	3	1	16.0	16.8

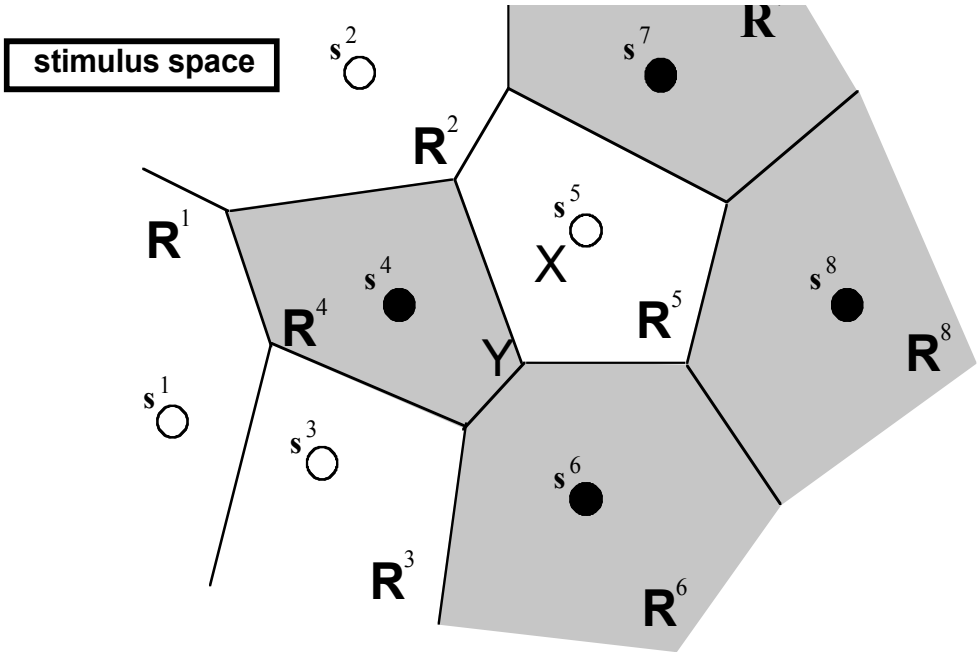


Figure 5.4. Voronoï partitioning. A set of 8 stored data points in a 2D stimulus space classified as either group **A** (filled circles) or **B** (open circles). Each stored point $\mathbf{s}^i$ "claims" a region of stimulus

space defined as the set of points that that are closer to $\mathbf{s}^i$ than all other $\mathbf{s}^j$. Any stimulus point to be classified by the system is assigned the class of its region; thus, dark regions are **A** and light regions are **B**. The two test patterns from the example are shown: **X=**(1,4) and **Y**=(-2,0).

5.4 Hebbian Learning

In the following chapters, we will examine models of learning that each start with a "naïve" network (i.e. randomly generated weights) and modify the weights according to a dynamical system that expresses $\dot{w}_{ij}$ as a function of other network variables. The purpose of this section is to describe a fundamental neural network learning principle. Like the concept of Nearest Neighbor learning description above, it is meant to provide context for the discussions in the subsequent chapters.

In his classic book, *The Organization of Behavior* (1949), D. O. Hebb articulated an extremely important principle, which he calls "*A Neurophysiological Postulate*" (see Box). This postulate, which was probably not intended to be quantified, has nevertheless inspired a number of mathematical formulations for adjusting weight values as a function of unit activity. Thus, the "Hebbian modification rule" takes the form [5-1] where a_i (t) is the activity of unit i, w_{ij} is the weight from unit j to unit i, and η is the learning rate (time step).

$$\Delta w_{ij} = \eta a_i(t) a_j(t) \qquad [5\text{-}1]$$

Hebb's *Neurophysiological Postulate*

Directly quoted from Hebb (1949)

"When an axon of cell *A* is near enough to excite a cell *B* and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that *A*'s efficiency, as one of the cells firing *B* is increased".

This form [Eq. 5-1] of the learning rule can have some serious problems, since if the activities are always positive (as any neurobiologist will tell you they are), then the weights will increase without bound. This divergence problem can be circumvented by suspending this constraint (for the moment, ignoring neurobiological dogma) and permit negative values of the activities (a_i). A small modification gets around the divergence problem:

$$\Delta w_{ij} = \eta (a_i(t) - \bar{a}_i)(a_j(t) - \bar{a}_j) \qquad [5\text{-}2]$$

This version of the Hebb rule insures that each factor has zero mean. Note that if the activities of two particular neurons are correlated (both tending to be above or below average at the same time), the weight between them will tend to increase, and if they are anti-correlated, the weight will decrease. Thus, a weight becomes a measure of the

correlation between the units it connects. In fact, if we begin with a weight value of zero, the weight array will become exactly equal to the covariance matrix **C**:

$$C_{ij} = \int_T (a_i(t) - \bar{a})(a_j(t) - \bar{a})dt \qquad [5\text{-}3]$$

Thus, the Hebb rule combines the biological plausibility of local influences on synaptic modification with a mathematical function (the covariance), which turns out to be very powerful computationally.

Hebbian associative learning.

Many neural network models of associative memory have been based on Hebb's rule. Some of the most well-known examples of these are Kohonen's (1977), Hopfield's (1977) and Anderson's (1977). The general idea is illustrated by considering a set of units each receiving a common set of stimulus patterns; that is, there is a single layer of weights as in Section 2.3. Consider a finite set $\mathcal{A}$ of K pairs of vectors, each pair a target vector of responses $\mathbf{T}^{\alpha}$ to be "associated" with a corresponding stimulus vector $\mathbf{s}^{\alpha}$. If the initial weight values are assumed to be zero, and the learning rate η is set to 1, the Hebbian Modification Rule (5-1) gives the following weights:

$$w_{ij} = \sum_{\alpha=1}^{K} T_i^{\alpha} s_j^{\alpha}, \qquad [5\text{-}4]$$

where w_{ij} is the weight from stimulus component j to output unit i. This set of weights establishes a kind of "association" between each stimulus vector $\mathbf{s}^{\alpha}$ and the corresponding target vector $\mathbf{T}^{\alpha}$. The response vector $\mathbf{r}(t)$ to a stimulus $\mathbf{s}(t)$, which may or may not be a stimulus from $\mathcal{A}$, is thus a weighted sum of the target vectors, where each weighting factor is the dot product between the corresponding stimulus vector $\mathbf{s}^{\alpha}$ and the presented stimulus $\mathbf{s}(t)$.

$$r_i(t) = \sum_{j=1}^{N} w_{ij} s_j(t) = \sum_{j=1}^{N} \sum_{\alpha=1}^{K} T_i^{\alpha} s_j^{\alpha} s_j(t) = \sum_{\alpha=1}^{K} \left(\mathbf{s}^{\alpha} \cdot \mathbf{s}(t)\right) T_i^{\alpha} \qquad [5\text{-}5]$$

or

$$\mathbf{r}(t) = \sum_{\alpha=1}^{K} \left(\mathbf{s}^{\alpha} \cdot \mathbf{s}(t)\right) \mathbf{T}^{\alpha} \qquad [5\text{-}6]$$

This simple system acts as a kind of associative memory to the extent that the dot product $\mathbf{s}^{\alpha} \cdot \mathbf{s}^{\beta}$ is a measure of *similarity* between the vectors. The ideal situation is when all the stimulus vectors are *orthonormal*; that is, The vectors all have magnitude 1 (they are *normal*) and the dot product between any two different stimulus vectors is 0 (they are *orthogonal*):

Example 5.2. Orthonormal Stimuli. Generate a Hebbian associative matrix for the following set of stimulus-target pairs.

Item (α)	Stimulus	Target
1	$0 \quad 1/\sqrt{2} \quad -1/\sqrt{2}$	$1 \quad 1$
2	$0 \quad 1/\sqrt{2} \quad 1/\sqrt{2}$	$0 \quad 1$
3	$1 \quad 0 \quad 0$	$1 \quad 0$

The weight from the first stimulus component to the first response component is the computed by summing the products of the first stimulus vector components and the first target vector components over all the patterns:

$$w_{11} = s_1^1 T_1^1 + s_1^2 T_1^2 + s_1^3 T_1^3 = (0)(1) + (0)(0) + (1)(1) = 1$$

The weight from the second stimulus component to the first response component would be:

$$w_{12} = s_2^1 T_1^1 + s_2^2 T_1^2 + s_2^3 T_1^3 = \left(\frac{1}{\sqrt{2}}\right)(1) + \left(\frac{1}{\sqrt{2}}\right)(0) + (0)(1) = \frac{1}{\sqrt{2}}$$

etc.

The resulting matrix is thus given by

$$\mathbf{W} = \begin{bmatrix} 1 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \\ 0 & \sqrt{2} & 0 \end{bmatrix}$$

The response $\mathbf{r}^\alpha$ to each stimulus $\mathbf{s}^\alpha$ is now computed for comparison with its corresponding target $\mathbf{T}^\alpha$:

$$\mathbf{r}^1 = \mathbf{W}\mathbf{s}^1 = \begin{bmatrix} 1 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \\ 0 & \sqrt{2} & 0 \end{bmatrix} \begin{bmatrix} 0 \\ \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \qquad \mathbf{r}^2 = \mathbf{W}\mathbf{s}^2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \qquad \mathbf{r}^3 = \mathbf{W}\mathbf{s}^3 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Orthonormal stimuli: $\mathbf{s}^\alpha \cdot \mathbf{s}^\beta = \begin{cases} 1 & \alpha = \beta \quad \text{(Normal)} \\ 0 & \alpha \neq \beta \quad \text{(Orthogonal)} \end{cases}$ [5-7]

In the orthonormal case, the system exhibits "perfect" retrieval, in that the response to any stimulus from $\mathcal{A}$ is the corresponding target vector.

Some noteworthy properties of the system:

- If $\mathbf{s}(t)$ is close to any of the learned stimuli $\mathbf{s}_\alpha$, the response will be close to the associated target $\mathbf{T}_\alpha$.
- If $\mathbf{s}(t)$ is a combination of 2 of the learned stimulus vectors, $\mathbf{s}_\alpha$ and $\mathbf{s}_\beta$, the response will be a combination of the associated targets $\mathbf{T}_\alpha$ and $\mathbf{T}_\beta$.
- The orthonormal case is not as unrealistic as it might seem, since for high dimensional stimuli (N large), arbitrary vectors tend to be "nearly" orthogonal.

Thus, a rather direct mathematical implementation of Hebb's hypothesis is seen to exhibit compelling properties of associative memory. The next chapter presents the *delta rule*, which is substantially more powerful and only slightly less simple.

5.5 Exercises

1. Given the data set below, classify the probes (1,-2) and (3,0) with the Nearest Neighbor method using S^{dot}, S^{cos}, and S^{gauss}.

Stored Points		
α	$\mathbf{s}_\alpha$	T_α
1	2 2	0
2	5 -1	0
3	-3 -5	0
4	-7 3	1
5	2 -1	0

2. Consider the concept of **Voronoï** regions applied to the cosine similarity measure. Sketch the regions for the set of points in Exercise 1.

3. Use the Hebb rule to find a weight matrix that will associate the following pattern pairs, and compute the responses of the system to each of the learned stimuli. Also compute the response of the system to the novel stimulus $\mathbf{s}(t) = [1,0,-1,0]$, and explain the result in terms of the learned stimulus-target pairs.

Stimulus	Target
1 1 -1 -1	1 0 1
1 -1 1 -1	0 1 0
1 -1 -1 1	1 1 1

4. Derive an expression for the expected absolute value of the cosine of the angle between two random vectors in N dimensions in terms of N, where each component is independently generated from a uniform distribution on the interval [-1,1]. What happens in the limit as N gets larger? Interpret this result.

5. Discuss each of the following cognitive tasks with respect to the type of learning involved: to what extent are "rules" learned, exceptions to the rules are there, and to what extent is table-lookup (memorization of facts) used?

Task	Input	Output
Multiplication	x, y (e.g. 4, 12)	x times y (e.g. 48)
Generation of past tense	Present tense (e.g. "jump")	Past tense (e.g. "jumped")
Reading aloud	written text	speech
Color naming	visual perception of color (e.g. image of a meadow)	color name (e.g. green)