INSTITUTE FOR NEURAL COMPUTATION
University of California, San Diego
Institute for Neural Computation
La Jolla, CA 92093-0523
Technical Report #0403 2 July 2004 inc2.ucsd.edu

Perceptrons

Robert Hecht-Nielsen
Computational Neurobiology, Institute for Neural Computation, ECE Department,
University of California, San Diego, La Jolla, California 92093-0407 USA rh-n@ucsd.edu

This is a reissue of a monograph that has been used at UCSD in graduate courses (ECE-270 and BGGN-269) and an advanced undergraduate course (ECE-173) on neural networks. Technical corrections found by students and instructors have been incorporated up to the current date given below. Particular thanks to Dr. Syrus Nemat-Nasser who has pointed out multiple important corrections. This monograph is being made available as a pdf for free use in neural network courses, and for neural network research, around the world. The views and opinions expressed reflect those of the author (although some have undoubtedly changed in the seven years since this monograph was written). Although it does not have exercises, this monograph should be of use for learning this important subject. Notwithstanding the emergence of more popular variants of the perceptron, such as the support vector machine, the classic perceptron construct is still the gold standard for regression and, in my view, for classification. I hope this monograph is of use to you. Best wishes for the intellectual adventure upon which you are about to embark.

Robert Hecht-Nielsen
15 February 2005

Perceptrons

Robert Hecht-Nielsen

Abstract

One of the intellectual pillars of cognitive science is *connectionism*; a doctrine which holds that a great deal can be learned about cognition by building and studying functional models of cognitive processes. This monograph describes one of the primary tools of connectionism: the *perceptron* artificial neural network architecture, the definition of which was finally completed by cognitive scientists in 1985 after decades of slow development. Often referred to in recent years as the *multilayer perceptron*, this monograph reverts to the original appellation of its inventor. The perceptron architecture (including its variants) was the first (and remains the only known) effective mechanism for adaptively building a model of a functional mapping from a large number of input variables to a large number of output variables. Unifying and reorganizing much of the research of the past 15 years, this book presents a complete and self-contained introduction to the perceptron, including its basic mathematical theory (which is provided both to demonstrate that it is simple and to give readers the high confidence that comes with knowledge to full depth).

1. Functions as Cognitive Process Models

Many cognitive processes can be modeled as a mathematical function **f** mapping n real variables to m real variables[1]

$$\mathbf{f} : R^n \to R^m$$
$$\mathbf{x} \mapsto \mathbf{y} = \mathbf{f}(\mathbf{x})$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ and $\mathbf{y} = (y_1, y_2, \ldots, y_m)^T$ (all elements of Euclidean spaces are herein expressed as column vectors). The requirement that must be met is *consistency*: the process produces the same output when applied at different times to the same input. Since noise, variability, and error are usually present in real-world processes, such a model must usually be considered an idealization. A general framework for viewing real-world processes and deriving functional models for them is considered below in Section 3.

A simple example of a cognitive process model is a *pattern classification* system. Such a system can be viewed as a function mapping a vector of n *feature values* **x** (termed a *pattern*) to a *classification vector* **y**. Here, the feature values

$$x_1, x_2, \ldots, x_n$$

are viewed as quantitative sensory or other measurements that have been made of the object to be classified and **y** is a *one-out-of-m* code indicating the class to which the object belongs ($\mathbf{y} = (1, 0, \ldots, 0)^T$ to indicate that the object belongs to class 1, $\mathbf{y} = (0, 1, \ldots, 0)^T$ to indicate that the

[1] This article assumes the reader has knowledge of elementary mathematical analysis (e.g., [Browder, 1996]).

object belongs to class 2, and so forth, up through $\mathbf{y} = (0, 0, \ldots, 1)^T$ indicating that the object belongs to class m – the total number of object classes being considered).

A classic example of pattern classification is *character recognition* [Hecht-Nielsen, 1990; Devroye et al, 1996] (see Figure 1). In this problem, the starting point is an image of an alphabetic or numeric character and the end point is a decision as to which character it is. A separate mechanism for locating, segmenting, and centering each character in the image box from the surrounding document is assumed.

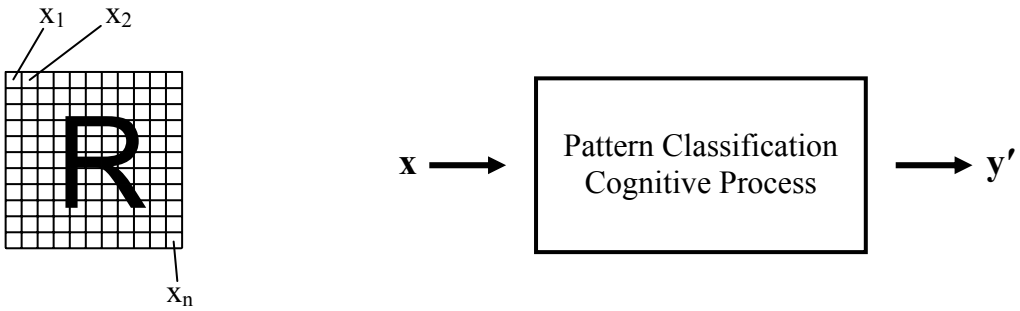

Figure 1. Character recognition. In this classic cognitive process model the input is an image of a centered, isolated alphabetic character (e.g., represented by a vector $\mathbf{x} = (x_1, x_2, ..., x_n)^T$ of image pixel grayscale brightness values; say, ranging from 0 – black to 255 – white). The output is a vector $\mathbf{y}' = (y'_1, y'_2, ..., y'_m)^T$ giving estimates of the a posteriori class membership probabilities. For example, in this **R** character case, the 'correct' output might be $\mathbf{y} = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0)^T$ (assuming that the classifier is only attempting to classify the 26 capital Latin alphabetic characters). In an operational setting the individual estimated class a posteriori probability values y'_j would be real numbers between 0.0 and 1.0. The output would, in this case, be considered correct if the value of y'_{18} (the component corresponding to an **R** character) were larger than all of the other y' components (this corresponds to the *Bayes' classification rule*, which is known to be optimal [Duda and Hart, 1973; Devroye et al, 1996]). The one-out-of-m codes are used because these average out to the class a posteriori probabilities required for optimal classifier accuracy [Haykin, 1999].

Creation of the character recognition system's capability is assumed to require a *training* process. During training, the system is supplied with a large number of example character images (e.g., as in Figure 1, vectors **x** which encode the pixel brightnesses of the image) and with the identity of the character shown in each image (in the form of a one-out-of-m classification vector **y**). These (**x**,**y**) examples are used to construct (e.g., using a perceptron, as discussed below) a *pattern classification* function cognitive model capable of mapping an image vector **x** into a class probability vector **y′** (where the apostrophe or 'prime' indicates that this vector is the output of the cognitive model). A functional model development process, such as this one, in which each input example **x** has an output **y** provided for it is termed *supervised training*. The ability of such a model to function well on new examples that it did not encounter during training is the key issue and is discussed extensively below.

A different example of a cognitive process model would be acquisition of machine control knowledge via supervised practice. For example, an automobile driving on a freeway might be equipped with sensors and sensor processing to produce a current *automobile state vector* **x** every few tens of milliseconds. The components of the state vector **x** would consist of data such as the total number of lanes on the road, the positions and velocities of all other cars within a half mile, the current road position and velocity of this automobile, the geometry of the road ahead, etc. Given the current state vector **x**, the model is to produce a current *command vector* **y′**, specifying the steering wheel angle and the forward (accelerator pedal) or backward (brake pedal) acceleration, which should be applied at this time. The model is invoked each time a new state vector is produced; thus updating the driving commands to the car every few tens of milliseconds.

Training of this cognitive model is carried out by having a human manually drive the instrumented car to generate example (**x**,**y**) input-output pairs for use in training the cognitive model. Following training, the cognitive model is tested by taking the car out on the road and letting the model drive (with a human safety driver). Cognitive models of this type are intended to explore acquisition and use of a type of knowledge which has been called *subsymbolic* (because it lies in a wide gray area between logic or rule-based knowledge and automatic control). A perceptron-based automobile driving system of this sort (for a simulated car driving in traffic on a freeway) has been built [Shepanski and Macy, 1988]. After training, this cognitive model demonstrated the ability to drive safely indefinitely in novel road scenarios, following the driving style of the human trainer. A real car of this type drove safely on freeways from Washington, DC to San Diego, CA.

The main point is that a wide variety of cognitive processes can be usefully modeled as functions. This is largely so because so many cognitive processes are, at least to a first approximation, deterministic and repeatable. Any process possessing these characteristics is automatically a function. Thus, in these instances, the use of functions as cognitive process models is scientifically, mathematically, and philosophically sound. As pointed up by the examples of character recognition and subsymbolic driving knowledge application, another important ability is to build our functional models using a 'training' or 'practice' paradigm. Sometimes, the characteristics of a cognitive model's gradual acquisition of knowledge during training are themselves interesting; as they might resemble, and provide possible insights into, aspects of human knowledge and skill acquisition.

2. Cognitive Process Data

In building a cognitive process model we assume the existence of a sufficiently large body of input-output examples $\{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_L, \mathbf{y}_L)\}$. The need for a sufficient number of examples is absolute. The example set must be large enough to statistically characterize the complete range of inputs and outputs. If the example set is not rich enough, the methods described in this book will not be mathematically sound. A practical test for determining whether enough data is available is described in Section 7 (a sometimes-effective method for synthesizing new data examples from a too-small body of real examples is also discussed there).

While there is no way around this absolute requirement for a sufficient quantity of data (either real or properly synthesized), what is surprising is that the actual number of examples needed is typically much smaller than might be intuitively supposed (this issue is also discussed in Section 7). If it is not possible to obtain (or synthesize) quite enough data, there exist *ill-posed problem methods* (incorporating user-supplied parameters imposing constraints on the model function such as specific limitations on it's derivatives) which can sometimes succeed in building a useful model function with slightly less data than is required to build a perceptron (see Section 7). If there is not enough data for even these methods, then it is time to give up and forget the whole thing. Any model which is built will probably not be meaningful.

After obtaining the example set, it is randomly divided into four randomly ordered disjoint subsets (random ordering is **crucial**). The first subset, called the *validation set*, is chosen to be large enough so that it will surely be highly representative of the input-output behavior of the system. The validation set is put aside and never examined or used until the very end. It is then used only once: to evaluate the performance of the final cognitive model. Only the results from such *sterile testing* are suitable for announcement or publication.

The second disjoint subset of data taken from the example set is the *training set*. This is typically the vast majority of the data. It is directly used for development of the cognitive model. The next is the *training test set*. As with the validation set, this is a statistically representative test set. However, this set is used many times during training to evaluate the performance of the model as it is developed (the elements of the training test set are never themselves directly used in developing the model; only to test its performance).

The final data subset used is the *verification set*. This set is used to carry out a final test of the cognitive model at the very end of its development (or of a few alternative models – see Section 7), just before the validation set is used. The purpose is to verify that the performance of the model using this set is approximately the same as that as measured using the training test set and that any performance goals have been met.

Once the verification set performance is deemed suitably consistent with the training test set performance, and the model meets any other performance requirements, it is subjected to final testing using the validation set. Once used for this testing, the validation set data must never be used again. Any reuse of this data in connection with the same modeling problem is fraudulent. The results obtained using the validation set give an excellent estimate of how the model actually performs in this statistical input-output environment and these results are suitable for communication to others.

Once the four data sets have been constructed, development of a cognitive model can begin.

3. Cognitive Processes and Noise

The use of a function as a cognitive process model is complicated by the ubiquitous presence of noise and error in the world. After decades of study and consideration of different formulations, an effective and widely applicable approach to mathematically formulating noise and error has emerged. The basic insight is to view the real-world data (upon which the cognitive capability is built) to be flawed, but to consider the resulting cognitive system itself as a deterministic function. This assumption automatically casts the solutions of relevant cognitive process modeling problems into being functions.

The basic question to be answered is which function the cognitive system should, ideally, be implementing. The most well developed approach is to assume that the function to be found is that one which implements the *average behavior* of the system being modeled. This concept, which has been successfully applied to a wide variety of cognitive process modeling problems, is now made precise.

The general formulation adopted here is that a randomly selected input-output pair $(\mathbf{x},\mathbf{y})$ from the particular cognitive process environment being considered can be viewed as a vector pair drawn at random with respect to a fixed (but unknown) non-negative *joint probability density* $\rho(\mathbf{x},\mathbf{y})$; which is assumed to be smooth (i.e., have continuous partial derivatives of all orders) and to be equal to zero outside a compact set with a smooth boundary (i.e., ρ has *compact support*). Of course, because $\rho(\mathbf{x},\mathbf{y})$ is a probability density, $\int_{R^m}\int_{R^n} \rho(\mathbf{x},\mathbf{y})\ d\mathbf{x}\, d\mathbf{y} = 1$.

Such a joint probability density $\rho(\mathbf{x},\mathbf{y})$ defines a *data source* -- a system which produces randomly selected input-output pairs $(\mathbf{x},\mathbf{y})$. The $\mathbf{x}$ vector portion of such a pair can be viewed as having been randomly chosen with respect to the *marginal probability density* $\rho(\mathbf{x}) \equiv \int_{R^m} \rho(\mathbf{x},\mathbf{y})\, d\mathbf{y}$. Given this

selected **x**, the **y** vector shall then be viewed as having been chosen at random with respect to the *conditional probability density* $\rho(\mathbf{y}\,|\,\mathbf{x})$, where $\rho(\mathbf{y}\,|\,\mathbf{x})$ is defined by $\rho(\mathbf{y}\,|\,\mathbf{x})\,\rho(\mathbf{x}) = \rho(\mathbf{x},\mathbf{y})$ (if $\rho(\mathbf{x}) = 0$ we define $\rho(\mathbf{y}\,|\,\mathbf{x}) \equiv 0$ for all **y**). Obviously, $\rho(\mathbf{x})$ and $\rho(\mathbf{y}\,|\,\mathbf{x})$ both have compact support because $\rho(\mathbf{x},\mathbf{y})$ does.

This smooth density-based formulation does not restrict generality, since any probability distribution restricted to a compact region (a class which includes all real-world situations as well as wild mathematical monstrosities) can be approximated to arbitrarily small mean squared error accuracy by such a density [Schmeisser and Triebel, 1987].

The *cognitive function* we are seeking to build shall therefore be assumed to be the vector average of the **y** output values which could occur in connection with a particular **x** input. This function is known as the *regression function* $\mathbf{r}(\mathbf{x}) \equiv \int_{R^m} \mathbf{y}\,\rho(\mathbf{y}\,|\,\mathbf{x})\,d\mathbf{y}$ of the data source $\rho(\mathbf{x},\mathbf{y})$ [Vapnik, 2000].

In constructing a cognitive process model it is important that the **x** and **y** vectors be carefully formulated so that the regression function will indeed be the function we are seeking. In particular, careful crafting of the **y** vector (so that the vector average of its noisy fluctuations will indeed be the function we want) is critical. Fortunately, the literature provides numerous and diverse examples of how this can be done (see Section 7 for references).

4. Perceptron

Constructing a regression function from randomly chosen noisy (**x**,**y**) data obtained from some system $\rho(\mathbf{x},\mathbf{y})$ would seem like a problem that would have been solved by statisticians long ago. Surprisingly, this is not so. Although many results were obtained by the 1950s for the case n = m = 1 (the single input variable / single output variable case), progress on the general high-dimensional case (n and m much larger than 1) was small until the 1980s. The breakthrough was the development of the *perceptron*, which, although it was anticipated by the early work of Rosenblatt [Rosenblatt, 1962] (also see Appendix A), who coined the term, and was originally discovered by others [Anderson and Rosenfeld, 1998], did not emerge to a wide audience until 1986 [Rumelhart et al, 1986; Rumelhart and McClelland, 1986]. The perceptron and its variants provide the only known effective general solution to the problem of building a regression function from examples in the high-dimensional input / high-dimensional output case. Many variants of the perceptron have been proposed and some of these (e.g., radial basis functions and support vector machines – see Appendix C) have also been extensively studied [Haykin, 1999; Vapnik, 2000; Fine, 1999; Devroye et al, 1996]. This book presents the standard, proven, baseline perceptron architecture. No attempt is made to describe or catalog its many variants.

Figure 2 below presents the general architecture of the perceptron. The *hidden neurons* of the perceptron carry out the calculation

$$z_j = \tanh\left(\sum_{k=0}^{n} u_{jk}\, x_k\right)$$

where j = 1, 2, …, M is the *index* of the hidden neuron and M is the number of hidden neurons in the perceptron. It is crucial to note that we define $x_0 \equiv z_0 \equiv 1.0$ (these are called *bias inputs* – and without

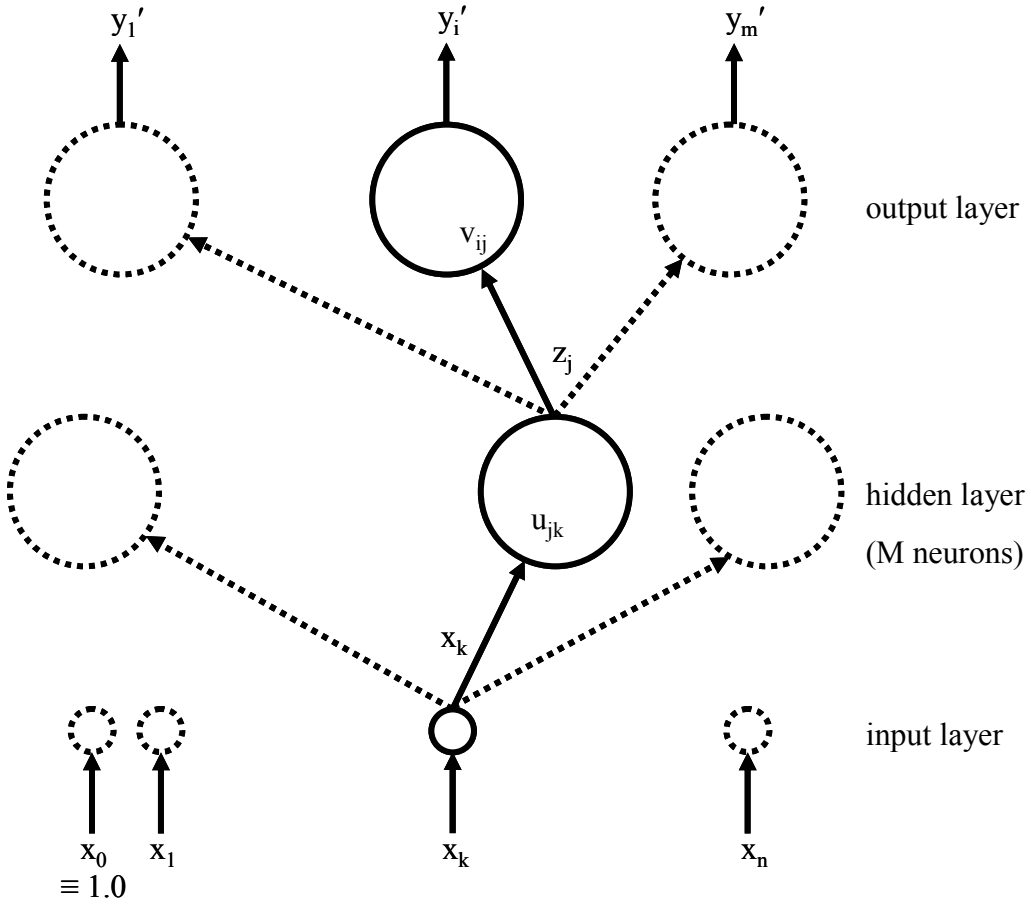

Figure 2. Perceptron. The input is an n-dimensional vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ and the output is an m-dimensional vector $\mathbf{y}' = (y_1', y_2', \ldots, y_m')^T$, where the primes indicate that this output is intended to approximate the output **r**(**x**) of the regression function. Each neuron of the middle (*hidden*) layer receives all of the components of the **x** vector (which are sent via the *input neurons* – which distribute these values). The output of each of the hidden neurons (the z_i's) is provided to each of the output neurons; which generate the y'_i's.

these, the perceptron is virtually useless[2]). After each hidden layer neuron has carried out its calculation and derived its z_j value, this value is sent as input to each of the *output layer neurons*. The output layer neurons then carry out the calculation

$$y_i' = \sum_{j=0}^{M} v_{ij}\, z_j$$

for output neuron index i = 1 to m. The final result is the m-dimensional output vector **y**′; which will often be expressed as **P**(**x**, M, **w**) (meaning perceptron output) where

$$\mathbf{w} = (w_1, w_2, \ldots, w_Q)^T \equiv (u_{10}, u_{11}, \ldots, u_{Mn}, v_{10}, v_{11}, \ldots, v_{mM})^T \qquad (1)$$

[2] The popular *PDP Books* [Rumelhart and McClelland, 1986], which described the modern perceptron in 1986, briefly mentioned, but did not emphasize, the need for bias inputs. For years afterwards it was common to meet people who complained that 'I tried the perceptron and it didn't work'. Almost always, the problem was a lack of bias inputs.

is the **weight vector** of the perceptron (Q is the total number of weights).

The first question that arises is: 'which functions can the perceptron implement?' The answer we want, of course, is 'all functions'. However, the definition of 'all' must first be provided. Since we are dealing with real-world situations, mathematical oddities such as infinitely jagged fractal functions can be dispensed with. Further, since $\rho(\mathbf{x},\mathbf{y})$ is assumed to have compact support, we need not consider unbounded domains or ranges. Excluding such possibilities, we shall consider only Riemann-integrable functions defined on a compact regular domain (which, when specificity is needed, will be assumed to be the unit hypercube $[0,1]^n$ – since any compact domain can be scaled and translated to fit within this cube).

A *finite harmonic polynomial* is a function of the form

$$\mathbf{H}(\mathbf{x}) = \sum_{|k_i| \le K} \mathbf{a}_\mathbf{k} \sin(2\pi\, \mathbf{k} \bullet \mathbf{x}) + \mathbf{b}_\mathbf{k} \cos(2\pi\, \mathbf{k} \bullet \mathbf{x})$$

where $\mathbf{k} = (k_1, k_2, \ldots, k_n)^T$ is an n-dimensional integer vector having $-K \le k_i \le K$ for all i, i = 1, 2, …, n; $\mathbf{a}_\mathbf{k}$ and $\mathbf{b}_\mathbf{k}$ are constant m-dimensional vectors; and $\bullet$ is the vector inner product. Note that the sum (which has a huge number of terms) is over all values of the vector index **k** with integer components less than or equal to K.

It is well known [Browder, 1996; Schmeisser and Triebel, 1987] that any Riemann-integrable function $\mathbf{f} : R^n \to R^m$ of compact support can be approximated to arbitrarily small mean squared (i.e., L_2 norm) error by harmonic polynomials[3]. Thus, any real-world function can be assumed to be a finite harmonic polynomial. A key question is whether a perceptron can also approximate any such function **f** to arbitrarily small mean squared error. The answer is provided by:

Theorem 1 [Universal Approximation Theorem]: Given any $\varepsilon > 0$ and any Riemann-integrable function $\mathbf{f} : [0,1]^n \subset R^n \to R^m$, there exists a perceptron **P**(**x**, M, **w**) such that

$$\int_{[0,1]^n} \left| \mathbf{f}(\mathbf{x}) - \mathbf{P}(\mathbf{x}, M, \mathbf{w}) \right|^2 d\mathbf{x} < \varepsilon \quad .$$

Proof: Since finite harmonic polynomials are universal approximators for Riemann-integrable functions, it will be sufficient to show that, given a sufficiently accurate finite harmonic polynomial **H**(**x**) approximation to **f**, we can find a perceptron **P**(**x**, M, **w**) which is a uniformly accurate approximation of **H**(**x**). In other words, for any $\gamma > 0$, we will show that there exists a perceptron **P**(**x**, M, **w**) for which $|\mathbf{H}(\mathbf{x}) - \mathbf{P}(\mathbf{x}, M, \mathbf{w})| < \gamma$ for every $\mathbf{x} \in [0,1]^n$.

Consider any term in the harmonic polynomial **H**(**x**); say $\mathbf{a}_\mathbf{k} \sin(2\pi\, \mathbf{k} \bullet \mathbf{x})$. Create a subcollection of J hidden neurons in the approximating perceptron which have their non-bias weights set as follows

$$u_{jp} = \alpha_j k_p \,,$$

[3] The class of functions that can be so approximated is actually larger, but the class of Riemann integrable functions is more than large enough for our purposes. This class includes, for example, all functions which are piecewise continuous on a finite number of Riemann-measurable regions making up their domains.

where j = 1, 2, … , J and p = 0, 1, 2, … , n. As shown in Figure 3, given any $\delta > 0$, by setting the α_j, real parameters β_j, and the u_{j0} weights correctly, the sum of the outputs of these hidden neurons,

$$\sum_{j=1}^{J} \beta_j \tanh(\alpha_j \left[2 \pi \mathbf{k} \bullet \mathbf{x}\right] + u_{j0})$$

will uniformly approximate $\sin(2 \pi \mathbf{k} \bullet \mathbf{x})$ to within absolute error δ for all $\mathbf{x} \in [0,1]^n$. This is because, with these proper choices of the u_{jp}, this sum forms a stairstep curve that will always stay within the $\sin(2 \pi \mathbf{k} \bullet \mathbf{x}) \pm \delta$ bounding curves over the entire **x** domain of the harmonic polynomial (the unit cube).

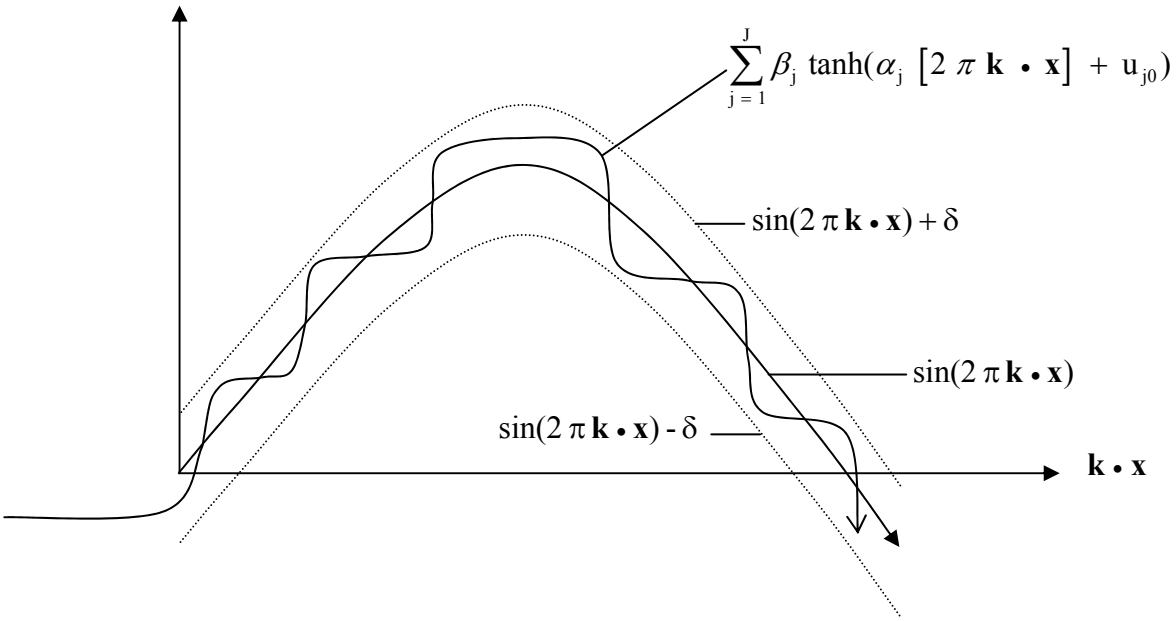

Figure 3. Uniform approximation of a harmonic polynomial term by a group of perceptron hidden neurons. As x varies over the unit cube $[0,1]^n$, the term's sine (or cosine – the argument is the same) function plots out sinusoidally as a function of $\mathbf{k} \bullet \mathbf{x}$. With the specific choices of the weights used, each hidden neuron in the group takes the form of a smooth step function. Given the proper values of the α_j's and u_{j0}'s, the sum of these hidden neuron outputs will form a multi-step curve which follows the sine function form of the polynomial term to within a uniform error of δ. No matter how small δ is chosen, this can be arranged. Thus, each term of a harmonic polynomial can be uniformly approximated to within this accuracy by a subgroup of hidden neurons.

If we now uniformly set all of the weights of each output neuron acting upon these J hidden neurons' inputs to the appropriate, β_j-adjusted, component of $\mathbf{a}_\mathbf{k}$ (i.e., to form the above sum of J hidden neuron outputs and then effectively multiply it by this component value), then the portion of each of the output neuron's output value attributable to these J hidden neurons will be equal, for each **x**, to the value of this component $\mathbf{a}_\mathbf{k} \sin(2 \pi \mathbf{k} \bullet \mathbf{x})$ of the harmonic polynomial term; plus or minus δ. The entire output of each output neuron will thus equal the corresponding vector component of the entire harmonic polynomial, plus or minus the product of the number of terms in the polynomial times δ. By choosing δ small enough, the desired overall uniform absolute approximation to within ε is achieved. ∎

Thus, perceptrons are *universal approximators*. Any real-world function can be approximated by a perceptron to an arbitrarily small mean squared error. However, given that harmonic polynomials are already universal approximators, why bother with perceptrons? The answer is that the number of terms

(hidden units) required is almost always a tiny fraction of the number of terms in a harmonic polynomial of equal accuracy. This is because the perceptron has an effective number of degrees of approximating freedom equal roughly to the product of the numbers of hidden and output layer weights; whereas the harmonic series has only as many degrees as freedom as there are individual coefficients in all the α_k and β_k vectors. Since the number of training examples required to construct a regression function approximation scales roughly linearly with the number of unknown parameters to be fitted, the dramatically smaller perceptron structures are practically useful, whereas harmonic polynomials are not. These factors probably largely account for the enormous practical success of the perceptron.

Finally, it is important to note that two perceptrons with different weight vectors and/or different numbers of hidden neurons can be *equivalent*. By equivalent it is meant that they have the same input-output behavior (given any **x** input supplied to both of them, the same **y**′ output vector always results from each). This property arises for several reasons. First, because tanh is an odd function, if we multiply all of the weights of a hidden neuron by –1 and then multiply all of the output neuron weights acting on this hidden neuron's output by –1, no change in input-output behavior of the perceptron will occur. Another situation occurs if extra hidden neurons are added which have all of their weights (and/or the output neuron weights acting upon their outputs) set to zero. Finally, because the output neuron transfer functions are affine, if two hidden neurons are permuted, and the corresponding weights which act on their outputs in the output neurons are permuted in the same way, there is no change in the overall output of the neural network.

An *irreducible* perceptron is a perceptron with one or more hidden neurons which possesses the following properties: there is exactly one hidden neuron with all-zero weights (the *zero neuron*), there are no two hidden neurons with the same weights, all hidden neurons have bias input weights (*bias weights*) which are non-negative, no hidden neuron has zero weights acting upon its output in all output neurons (other than the zero neuron, which **must** have all zero weights acting upon its output), and the bias weight of each hidden neuron is greater than or equal to that of the hidden neuron immediately to its right (if two or more hidden neurons have the same bias weight value then they are ordered in descending order to the right with respect to their subsequent weights). A *constant perceptron* is an irreducible perceptron with only one hidden neuron (the zero neuron). The *null perceptron* is the constant perceptron having all output neuron bias weights equal to zero. It can be shown that every perceptron is equivalent to one and only one irreducible perceptron [Sussmann, 1992; Chen, Lu, and Hecht-Nielsen, 1993; Kůrkova and Kainen, 1994].

5. Mean Squared Error

Given a data source $\rho(\mathbf{x},\mathbf{y})$, our ultimate goal is to find a way to implement its regression function $\mathbf{r}(\mathbf{x})$. To proceed, the results of this section will be required.

Let $\mathbf{f}(\mathbf{x})$ be any Riemann-integrable function from R^n to R^m. The *mean squared error* $F[\mathbf{f}]$ of this function (as an approximation to the input-output pairs $(\mathbf{x},\mathbf{y})$ which might occur) is defined to be

$$F[\mathbf{f}] \equiv \int_{R^n} \int_{R^m} | \mathbf{y} - \mathbf{f}(\mathbf{x}) |^2 \, \rho(\mathbf{x}, \mathbf{y}) \, d\mathbf{y} \, d\mathbf{x} .$$

Using the fact that, for vectors **a** and **b**, $| \mathbf{a} + \mathbf{b} |^2 = | \mathbf{a} |^2 + 2 \, \mathbf{a} \cdot \mathbf{b} + | \mathbf{b} |^2$, the mean squared error can then be re-expressed using the following steps:

$$F[\mathbf{f}] = \int_{R^n} \int_{R^m} |\mathbf{y} - \mathbf{r}(\mathbf{x}) + \mathbf{r}(\mathbf{x}) - \mathbf{f}(\mathbf{x})|^2 \, \rho(\mathbf{x}, \mathbf{y}) \, d\mathbf{y} \, d\mathbf{x}$$

$$= \int_{R^n} \int_{R^m} \left(|\mathbf{y} - \mathbf{r}(\mathbf{x})|^2 + 2[\mathbf{y} - \mathbf{r}(\mathbf{x})] \bullet [\mathbf{r}(\mathbf{x}) - \mathbf{f}(\mathbf{x})] + |\mathbf{r}(\mathbf{x}) - \mathbf{f}(\mathbf{x})|^2 \right) \rho(\mathbf{x}, \mathbf{y}) \, d\mathbf{y} \, d\mathbf{x}$$

$$F[\mathbf{f}] = \int_{R^n} \int_{R^m} |\mathbf{y} - \mathbf{r}(\mathbf{x})|^2 \, \rho(\mathbf{x}, \mathbf{y}) \, d\mathbf{y} \, d\mathbf{x} + 2 \int_{R^n} \int_{R^m} [\mathbf{y} - \mathbf{r}(\mathbf{x})] \bullet [\mathbf{r}(\mathbf{x}) - \mathbf{f}(\mathbf{x})] \, \rho(\mathbf{x}, \mathbf{y}) \, d\mathbf{y} \, d\mathbf{x}$$

$$+ \int_{R^n} \int_{R^m} |\mathbf{r}(\mathbf{x}) - \mathbf{f}(\mathbf{x})|^2 \, \rho(\mathbf{x}, \mathbf{y}) \, d\mathbf{y} \, d\mathbf{x}$$

$$F[\mathbf{f}] = \int_{R^n} \int_{R^m} |\mathbf{y} - \mathbf{r}(\mathbf{x})|^2 \, \rho(\mathbf{x}, \mathbf{y}) \, d\mathbf{y} \, d\mathbf{x} + 2 \int_{R^n} [\mathbf{r}(\mathbf{x}) - \mathbf{f}(\mathbf{x})] \bullet \left[\int_{R^m} [\mathbf{y} - \mathbf{r}(\mathbf{x})] \, \rho(\mathbf{x}, \mathbf{y}) \, d\mathbf{y} \right] d\mathbf{x}$$

$$+ \int_{R^n} |\mathbf{r}(\mathbf{x}) - \mathbf{f}(\mathbf{x})|^2 \left(\int_{R^m} \rho(\mathbf{x}, \mathbf{y}) \, d\mathbf{y} \right) d\mathbf{x}$$

The first of these latter three integrals is a non-negative constant which depends only upon $\rho(\mathbf{x},\mathbf{y})$ and not upon $\mathbf{f}$. Call it A. The third integral can be further simplified by using the marginal density definition. Expanding the second integral further using the definition of the conditional density gives

$$F[\mathbf{f}] = A + 2 \int_{R^n} [\mathbf{r}(\mathbf{x}) - \mathbf{f}(\mathbf{x})] \bullet \left[\int_{R^m} [\mathbf{y} - \mathbf{r}(\mathbf{x})] \, \rho(\mathbf{y} \,|\, \mathbf{x}) \, \rho(\mathbf{x}) \, d\mathbf{y} \right] d\mathbf{x}$$

$$+ \int_{R^n} |\mathbf{r}(\mathbf{x}) - \mathbf{f}(\mathbf{x})|^2 \, \rho(\mathbf{x}) \, d\mathbf{x}$$

When the definition of the regression function is invoked, this last expression yields

$$F[\mathbf{f}] = A + 2 \int_{R^n} [\mathbf{r}(\mathbf{x}) - \mathbf{f}(\mathbf{x})] \bullet [\mathbf{r}(\mathbf{x}) - \mathbf{r}(\mathbf{x})] \, \rho(\mathbf{x}) \, d\mathbf{x} + \int_{R^n} |\mathbf{r}(\mathbf{x}) - \mathbf{f}(\mathbf{x})|^2 \, \rho(\mathbf{x}) \, d\mathbf{x}$$

$$= A + \int_{R^n} |\mathbf{r}(\mathbf{x}) - \mathbf{f}(\mathbf{x})|^2 \, \rho(\mathbf{x}) \, d\mathbf{x}$$

Thus, the following result has been established.

Theorem 2: The mean squared error $F[\mathbf{f}]$

$$F[\mathbf{f}] \equiv \int_{R^n} \int_{R^m} |\mathbf{y} - \mathbf{f}(\mathbf{x})|^2 \, \rho(\mathbf{x}, \mathbf{y}) \, d\mathbf{y} \, d\mathbf{x}$$

of a function $\mathbf{f}(\mathbf{x})$ with respect to a data source $\rho(\mathbf{x},\mathbf{y})$ with regression function $\mathbf{r}(\mathbf{x}) \equiv \int \mathbf{y} \, \rho(\mathbf{y} \,|\, \mathbf{x}) \, d\mathbf{y}$ can be written as

$$F[\mathbf{f}] = A + \int_{R^n} |\mathbf{r}(\mathbf{x}) - \mathbf{f}(\mathbf{x})|^2 \rho(\mathbf{x}) \, d\mathbf{x},$$

where A is a non-negative constant called the *variance* of $\rho(\mathbf{x},\mathbf{y})$ (because it is the average squared magnitude of deviation of **y** around its average value $\mathbf{r}(\mathbf{x})$). ■

Thus, if we want to find the regression function, we need to find the function **f** which minimizes F[**f**].

It is important to note that the mean squared error is highly sensitive to the relative magnitudes of the components of the **y** vectors (and, given the perceptron approach introduced in the next section, of the **x** vectors as well). For example, if one component of **y** is generally a million times larger in magnitude than a second component, the effect of errors in the first may completely overshadow errors in the second. A method based upon minimizing mean squared error might then spend all of its initial effort minimizing errors in the larger-scale variables; ignoring smaller-scale variables until this larger error source has been adequately addressed (a process which, in a practical method, might require more than a human lifetime to complete).

In effect, the relative sizes of the mean squared errors in the different components of **y** determine their relative *importance* with respect to influence on reduction of overall mean squared error by some sample error minimization (i.e., learning) procedure. What we usually strive for is a formulation of the **y′** vectors such that each component has roughly equal mean squared error and equal *importance* to the problem at hand (i.e., we care equally about minimizing the mean squared errors of all of the **y′** components, and similarly for **x**). Transformation of raw (**x**,**y**) data vectors into such a desirable form is called *prescaling* (since it is done prior to any use of the data). Many effective prescaling techniques have been developed, and while these are often among the most important ingredients in perceptron application success, they are too problem-dependent for consideration here. However, there is a crude prescaling technique which is almost always used as a first step (and sometimes suffices) – *z-scaling*.

z-scaling works by computing the mean and variance of each component of both the **x** and the **y** vectors across large numbers of examples and then subtracting the mean and dividing by the standard deviation (the square root of the variance). Specifically, given a large set $\{(\mathbf{x}_1,\mathbf{y}_1),(\mathbf{x}_2,\mathbf{y}_2),\ldots,(\mathbf{x}_L,\mathbf{y}_L)\}$ of examples, chosen at random with respect to $\rho(\mathbf{x},\mathbf{y})$, we compute the following quantities:

$$\mu_i^x = \frac{1}{L}\sum_{k=1}^{L} x_{ki} \qquad \text{for } i = 1, 2, \ldots, n$$

$$\mu_j^y = \frac{1}{L}\sum_{k=1}^{L} y_{kj} \qquad \text{for } j = 1, 2, \ldots, m$$

$$\sigma_i^x = \sqrt{\frac{1}{L}\sum_{k=1}^{L} (x_{ki} - \mu_i^x)^2} \qquad \text{for } i = 1, 2, \ldots, n$$

$$\sigma_j^y = \sqrt{\frac{1}{L}\sum_{k=1}^{L} (y_{kj} - \mu_j^y)^2} \qquad \text{for } j = 1, 2, \ldots, m \ .$$

Once these are calculated and frozen (they are calculated only once at the outset and then never changed), the original data set $\{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_L, \mathbf{y}_L)\}$ (now referred to as $\{(\mathbf{x}_1^{\text{original}}, \mathbf{y}_1^{\text{original}}), (\mathbf{x}_2^{\text{original}}, \mathbf{y}_2^{\text{original}}), \ldots, (\mathbf{x}_L^{\text{original}}, \mathbf{y}_L^{\text{original}})\}$) is converted to a new data set $\{(\mathbf{x}_1^{\text{scaled}}, \mathbf{y}_1^{\text{scaled}}), (\mathbf{x}_2^{\text{scaled}}, \mathbf{y}_2^{\text{scaled}}), \ldots, \mathbf{x}_L^{\text{scaled}}, \mathbf{y}_L^{\text{scaled}})\}$ to be used in building the perceptron approximation to the regression function. Applying the following z-scaling transformations to the vector components carries out this conversion process from each $(\mathbf{x}^{\text{original}}, \mathbf{y}^{\text{original}})$ to the corresponding $(\mathbf{x}^{\text{scaled}}, \mathbf{y}^{\text{scaled}})$

$$x_i^{\text{scaled}} = \frac{x_i^{\text{original}} - \mu_i^{\mathbf{x}}}{\sigma_i^{\mathbf{x}}} \qquad \text{for } i = 1, 2, \ldots, n$$

$$y_j^{\text{scaled}} = \frac{y_j^{\text{original}} - \mu_j^{\mathbf{y}}}{\sigma_j^{\mathbf{y}}} \qquad \text{for } j = 1, 2, \ldots, m \ .$$

In all matters involved with creating an implementation of the regression function (subdividing the data into the training set, training test set, etc.; testing the implementation at various points during development; and so forth) the scaled data is used exclusively. However, when it is desired to use the perceptron cognitive model which has been created, it is necessary to scale incoming data (in accordance with the above z-scaling equations for **x**) and then un-scale the outputs of the perceptron using the reverse of the z-scaling equations for **y**, which are given by

$$y_j^{\text{original}} = \sigma_j^{\mathbf{y}} \, y_j^{\text{scaled}} + \mu_j^{\mathbf{y}} \qquad \text{for } j = 1, 2, \ldots, m \ .$$

Finally, it is important to note that z-scaling does not always work as a prescaling method. This is because it is only effective when the components of x and y have roughly *linearly-scaled behavior*; meaning that a fixed change in any component has roughly the same 'meaning' and 'importance,' independent of the component's value. An example of where this would not hold would be where a component was light intensity and the model is considering human perceptual response. Since human response to light is logarithmic, a change of 10 units of intensity when the intensity is 10 is vastly more meaningful than when the intensity is 10,000,000. In this case, first replacing the light intensity with its logarithm and then z-scaling solves the problem.

Once prescaling has been carried out, the final step is to look for *erroneous outliers* in all of the prescaled data. *Outliers* are $(\mathbf{x}^{\text{scaled}}, \mathbf{y}^{\text{scaled}})$ points that have components with values much larger in magnitude than 1.0. Such points should be translated back into their raw $(\mathbf{x}^{\text{original}}, \mathbf{y}^{\text{original}})$ form and, if possible, be individually examined to see if they make sense. Often, these unusually large components are determined to be erroneous (mistakes, data corruptions, etc.). Erroneous outliers (and other erroneous data points, if they can be found and confirmed) should be discarded from all four data sets, as they can cause great problems during the perceptron development and testing process.

Clearly, prescaling is an art. However, in practice it is rarely difficult to find workable solutions to prescaling difficulties. In what follows, it will always be assumed that successful prescaling has been carried out (meaning that roughly linearly-scaled behavior has been achieved in all the components of both the **x** and the **y** vectors and that z-scaling has then been carried out). The only exception is when

pattern classification is the function being carried out; in which case only the **x** vectors are prescaled (the **y** vectors – the one-out-of-m pattern class codes – are left alone).

6. Perceptron Learning

This section discusses the process of building a perceptron to approximately implement the regression function $\mathbf{r}(\mathbf{x})$ of a data source $\rho(\mathbf{x},\mathbf{y})$. This *learning* process is based upon using a training set consisting of randomly chosen (and successfully prescaled! -- this won't be mentioned again) examples from the source.

To start with, since, by Theorem 1, perceptrons are universal approximators of real-world functions, we can assume, without loss of generality, that the regression function $\mathbf{r}(\mathbf{x})$ of the data source $\rho(\mathbf{x},\mathbf{y})$ we are considering is itself a perceptron (say, placed in irreducible form, with $M_r \geq 2$ hidden neurons and weight vector $\mathbf{w_r}$)[4]. In other words, we assume that $\mathbf{r}(\mathbf{x}) \equiv \mathbf{P}(\mathbf{x}, M_r, \mathbf{w_r})$. Further, we assume that the function **f** that we seek is also a perceptron. Call it $\mathbf{P}(\mathbf{x}, M, \mathbf{w})$ (unlike the regression function perceptron, this perceptron is not assumed to be in irreducible form). Thus, the mean squared error can be written as

$$F(M,\mathbf{w}) = A + \int_{R^n} \left| \mathbf{P}(\mathbf{x}, M_r, \mathbf{w_r}) - \mathbf{P}(\mathbf{x}, M, \mathbf{w}) \right|^2 \rho(\mathbf{x})\ d\mathbf{x}\ .$$

However, it is easy to see that the difference of two perceptrons is itself a perceptron. Call this difference perceptron $\mathbf{S}(\mathbf{x}, M, \mathbf{w})$. So, we can write[5]:

$$F(M,\mathbf{w}) = A + \int_{R^n} \left| \mathbf{S}(\mathbf{x}, M, \mathbf{w}) \right|^2 \rho(\mathbf{x})\ d\mathbf{x}$$

Clearly, our goal is to find M and **w** so that **S** is the zero perceptron. This implies that $\mathbf{P}(\mathbf{x}, M_r, \mathbf{w_r})$ and $\mathbf{P}(\mathbf{x}, M, \mathbf{w})$ are equivalent. This can only happen if $M \geq M_r$, since $\mathbf{P}(\mathbf{x}, M_r, \mathbf{w_r})$ is irreducible. Thus, if $M < M_r$, then **S** cannot be equivalent to the zero perceptron. In this case, $F(M, \mathbf{w})$ will be strictly greater than A.

If $M \geq M_r$, then **w** vectors will always exist such that **S** is the zero perceptron (these are not unique -- call any such a pair of values M^* and $\mathbf{w}^*$). Thus, in this case, $F(M^*,\mathbf{w}^*)$ will be equal to A.

For any fixed M (we shall assume M is fixed in most of what follows – procedures for fixing it are discussed in the next section), the mean squared error function $F(M, \mathbf{w})$ can be viewed as defining the non-negative *altitude* or *height* of a surface hovering over the **w** weight space. This surface (which is smooth because the perceptron is a smooth function of its weights) is known as the *error surface* of the density $\rho(\mathbf{x},\mathbf{y})$.

---

[4] If $M_r = 1$ then the regression function is a constant perceptron – an uninteresting case.

[5] Statisticians call a statistical quantity such as $F(M,\mathbf{w})$ *robust* if it is not subject to *breakdown* (a huge variation in the optimum parameters M and **w** when a small change in $\rho(\mathbf{x},\mathbf{y})$ occurs) [Huber, 1996]. Sadly, $F(M,\mathbf{w})$ is not robust. Nonetheless, so many practical problems can be well-solved using $F(M,\mathbf{w})$ that these theoretical shortfalls do not seem to be important.

Because the error surface is smooth, given any point $\mathbf{w}_0$, we can expand the mean squared error function in a finite exact multidimensional Taylor series truncated at three terms (see [Browder, 1996])

$$F(M, \mathbf{w}_0 + \mathbf{h}) = F(M, \mathbf{w}_0) + \left[\nabla_{\mathbf{w}} F(M, \mathbf{w})|_{\mathbf{w}=\mathbf{w}_0}\right] \cdot \mathbf{h} + \frac{1}{2}\mathbf{h}^T \left[\frac{\partial F(M, \mathbf{w})}{\partial w_i \partial w_j}\right]_{\mathbf{w}=\mathbf{w}_0+\theta\mathbf{h}} \mathbf{h}, \quad (2)$$

where the Q x Q matrix in the third term is the *Hessian* H and where $0 < \theta < 1$ depends, in general, on $\mathbf{w}_0$ and $\mathbf{h}$. From this formula it is easy to see that, as long as $\nabla_{\mathbf{w}} F(M, \mathbf{w})|_{\mathbf{w}=\mathbf{w}_0} \neq \mathbf{0}$, we can set $\mathbf{h} = -\alpha \nabla_{\mathbf{w}} F(M, \mathbf{w})|_{\mathbf{w}=\mathbf{w}_0}$, $\alpha > 0$, and then get

$$F(M, \mathbf{w}_0 + \mathbf{h}) = F(M, \mathbf{w}_0) - \alpha \left|\nabla_{\mathbf{w}} F(M, \mathbf{w})|_{\mathbf{w}=\mathbf{w}_0}\right|^2 + \frac{1}{2}\mathbf{h}^T H(M, \mathbf{w}_0 + \theta \mathbf{h})\mathbf{h}. \quad (3)$$

Since F is a smooth function, its gradient $\nabla_{\mathbf{w}} F(M, \mathbf{w})$ and its Hessian H are also both smooth. Further, we can assume that the weight vectors are limited to some compact set (e.g., the set of all weight values that lie within the floating point range of a computer).

Thus, both the gradient and Hessian of F are bounded (the magnitude $|\nabla_{\mathbf{w}} F(M, \mathbf{w})|$ is bounded and the norm $\|H(M, \mathbf{w})\| \equiv \sup_{|\hat{\mathbf{h}}| \leq 1} \left[\left|\hat{\mathbf{h}}^T H(M, \mathbf{w}) \hat{\mathbf{h}}\right|\right]$ (where the supremum is taken over all unit vectors $\hat{\mathbf{h}}$) is bounded for all $\mathbf{w}$'s within our compact set). Thus, if we re-write Equation 3 as

$$F(M, \mathbf{w}_0 + \mathbf{h}) \leq F(M, \mathbf{w}_0) - \alpha \left|\nabla_{\mathbf{w}} F(M, \mathbf{w})|_{\mathbf{w}=\mathbf{w}_0}\right|^2 + \frac{1}{2}\alpha^2 \|H(M, \mathbf{w})\|$$

it is clear that we can find α small enough so that the third term is smaller in magnitude than the second (this is because, for sufficiently small α, α^2 can be as small as we wish in comparison to α). In fact, because the Hessian is bounded, if, for some small constant $c > 0$, we only consider $\mathbf{w}_0$'s such that that $\left|\nabla_{\mathbf{w}} F(M, \mathbf{w})|_{\mathbf{w}=\mathbf{w}^{old}}\right| \geq c$, then we can find a value of α that will work uniformly at **every** such $\mathbf{w}_0$. Thus, we have proved

Theorem 3: Given any point $\mathbf{w}^{old}$ such that $\nabla_{\mathbf{w}} F(M, \mathbf{w})|_{\mathbf{w}=\mathbf{w}_0} \neq \mathbf{0}$, there exists a positive number α_0 such that for all $0 < \alpha \leq \alpha_0$ the value of $F(M, \mathbf{w}^{new})$ is strictly smaller than the value of $F(M, \mathbf{w}^{old})$; where

$$\mathbf{w}^{new} = \mathbf{w}^{old} - \alpha \nabla_{\mathbf{w}} F(M, \mathbf{w})|_{\mathbf{w}=\mathbf{w}^{old}}. \quad (4)$$

Further, if we assume that the weight vector $\mathbf{w}$ is restricted to a compact set and, for some positive constant c, consider only those points $\mathbf{w}^{old}$ for which $\left|\nabla_{\mathbf{w}} F(M, \mathbf{w})|_{\mathbf{w}=\mathbf{w}^{old}}\right| \geq c$ (such a subset of R^Q is called *admissible*), then there exists a fixed α_0 such that for all $\mathbf{w_0}$ in this set the value for F will strictly decrease for all $0 < \alpha < \alpha_0$.

If we apply the process of Equation 4 repeatedly, starting at some initially selected weight vector $\mathbf{w}^{old}$, we will keep reducing the mean squared error. In other words, this *gradient descent perceptron learning process* creates a series of ever-better weight vectors. Hopefully, by continuing this process, we will eventually end up at a weight vector very close to a $\mathbf{w}^*$.

In order to implement the gradient descent learning process we need two things: an initial $\mathbf{w}^{old}$ weight vector and a way of calculating the gradient $\nabla_{\mathbf{w}} F(M, \mathbf{w})|_{\mathbf{w}=\mathbf{w}^{old}}$ at each step of the process. These two issues are now discussed.

Selecting a good initial $\mathbf{w}^{old}$ weight vector is important. This is because of two things. First, the initial weight vector determines the initial functional form of the perceptron. If any of the hidden layer weights are 'large' (relative to the prescaled **x** and **y** component variance range of [-1, +1]), the derivatives of the hyperbolic tangent functions will be very small. This means that the magnitudes of the corresponding components of $\nabla_{\mathbf{w}} F(M, \mathbf{w})|_{\mathbf{w}=\mathbf{w}^{old}}$ will be very small, and **h** (the distance moved in weight space towards the goal $\mathbf{w}^*$) will be very small. This greatly slows down the progress of learning.

The other problem with large initial weight values is that the initial perceptron function is then a complicated and particular function – a decision which the learning process must then typically first 'undo' before making progress towards implementing the actual regression function. Thus, it is better to start off with a more parsimonious perceptron. The standard perceptron training procedure is to start off the weights at 'small' values chosen at random in accordance with a small-variance Gaussian or uniform probability density with mean zero.

For example, the initial value of each weight might be chosen at random using the Gaussian density G(0, 0.0001) with mean 0 and variance 0.0001 (standard deviation 0.01). With these small weight values the function implemented by the perceptron has an output which is approximately the zero vector **0** for all **x** inputs. In other words, each component function is essentially a 'flat surface'. As training proceeds, these initially flat surfaces are gradually deformed and molded by the learning process into the forms of the regression function component surfaces.

Given the above *learning law* (Equation 4) the main issue we must address is how to calculate the required gradient $\nabla_{\mathbf{w}} F(M, \mathbf{w})|_{\mathbf{w}=\mathbf{w}^{old}}$ at each step. To carry out this calculation we first re-express F(M, **w**) as

$$\begin{aligned} F(M, \mathbf{w}) &\equiv \int_{R^m} \int_{R^n} |\, \mathbf{y} - \mathbf{P}(\mathbf{x}, M, \mathbf{w}) \,|^2 \; \rho(\mathbf{x}, \mathbf{y}) \; d\mathbf{x}\, d\mathbf{y} \\ &\equiv \underset{\rho(\mathbf{x},\mathbf{y})}{E} \left[|\, \mathbf{y} - \mathbf{P}(\mathbf{x}, M, \mathbf{w}) \,|^2 \right] \\ &\overset{a.s.}{=} \lim_{N \to \infty} \frac{1}{N} \sum_{k=1}^{N} |\, \mathbf{y}_k - \mathbf{P}(\mathbf{x}_k, M, \mathbf{w}) \,|^2 \end{aligned} \tag{5}$$

where $\underset{\rho(\mathbf{x},\mathbf{y})}{E}[\;]$ means the expectation value or average of the quantity in brackets over all randomly chosen (**x**,**y**)'s and where a.s. (*almost surely*) indicates equality with probability one (a fine technical distinction that shall hereafter be ignored) and where the ($\mathbf{x}_k$, $\mathbf{y}_k$) are input-output examples chosen at random with respect to ρ(**x**,**y**). This third equality is established by the *Monte Carlo Theorem* (see Appendix B); and this limit converges rapidly. Thus, given a sufficiently large set of examples ($\mathbf{x}_k$, $\mathbf{y}_k$),

F(M, **w**) can be approximated by this rapidly converging sum to any desired degree of accuracy – a fact which will be used below.

Given this new expression for F(M, **w**), we now calculate $\nabla_{\mathbf{w}} F(M, \mathbf{w})|_{\mathbf{w}=\mathbf{w}^{old}}$. First, recall that

$$\nabla_{\mathbf{w}} F(M, \mathbf{w}) \equiv \left(\frac{\partial F(M, \mathbf{w})}{\partial w_1}, \frac{\partial F(M, \mathbf{w})}{\partial w_2}, \ldots, \frac{\partial F(M, \mathbf{w})}{\partial w_Q} \right)^T$$

where, again, the w_i's are (as per Equation 1) the individual weights of the perceptron. Thus, what we need to calculate is $\frac{\partial F(M, \mathbf{w})}{\partial w_i}$.

Because **x** and **y** are governed by a smooth density ρ(**x**,**y**) of compact support and because $\mathbf{P}(\mathbf{x}_k, M, \mathbf{w})$ is a smooth function of **w**, we can differentiate the last form of Equation 5 inside the limit. This gives us

$$\frac{\partial F(M, \mathbf{w})}{\partial w_i} = -2 \lim_{N \to \infty} \frac{1}{N} \sum_{k=1}^{N} \sum_{j=1}^{m} \left[y_{kj} - P_j(\mathbf{x}_k, M, \mathbf{w}) \right] \frac{\partial P_j(\mathbf{x}_k, M, \mathbf{w})}{\partial w_i} \quad , \tag{6}$$

where y_{kj} and $P_j(\mathbf{x}_k, M, \mathbf{w})$ are, respectively, the j^{th} components of the vectors $\mathbf{y}_k$ and $\mathbf{P}(\mathbf{x}_k, M, \mathbf{w})$, and where the fact that

$$\left| \mathbf{y}_k - \mathbf{P}(\mathbf{x}_k, M, \mathbf{w}) \right|^2 = \sum_{j=1}^{m} \left[y_{kj} - P_j(\mathbf{x}_k, M, \mathbf{w}) \right]^2$$

has been used. Obviously, the next step is calculation of $\frac{\partial P_j(\mathbf{x}_k, M, \mathbf{w})}{\partial w_i}$. This is now done for the two cases:

1) w_i is an output neuron weight v_{pq}

or

2) w_i is a hidden neuron weight u_{qr}.

Recall that

$$P_j(\mathbf{x}_k, M, \mathbf{w}) \equiv \sum_{\ell=0}^{M} v_{j\ell} \, z_\ell(\mathbf{x}_k, \mathbf{u}) \tag{7}$$

where the output $z_\ell(\mathbf{x}_k, \mathbf{u})$ of hidden neuron ℓ is written as a function of the input vector $\mathbf{x}_k$ and the vector of hidden neuron weights **u**, and where

$$\begin{aligned} z_0(\mathbf{x}_k, \mathbf{u}) &\equiv 1.0 \\ z_\ell(\mathbf{x}_k, \mathbf{u}) &\equiv \tanh\left(\sum_{s=0}^{n} u_{\ell s} x_{ks} \right) , \quad \text{for } \ell > 0 \end{aligned} \tag{8}$$

and

$$x_{ko} \equiv 1.0 .$$

Case 1: Assume $w_i = v_{pq}$ – the weight of output neuron p ($1 \le p \le m$) which acts upon the input z_q from hidden neuron q ($0 \le q \le M$).

Then

$$\frac{\partial P_j(\mathbf{x}_k, M, \mathbf{w})}{\partial w_i} \equiv \frac{\partial P_j(\mathbf{x}_k, M, \mathbf{w})}{\partial v_{pq}} = \begin{cases} 0 & \text{if } p \neq j \\ z_q(\mathbf{x}_k, \mathbf{u}) & \text{if } p = j \end{cases}$$

Case 2: Assume $w_i = u_{qr}$ – the weight of hidden neuron q which acts upon input x_r, then, using Equations 7 and 8,

$$\frac{\partial P_j(\mathbf{x}_k, M, \mathbf{w})}{\partial w_i} \equiv \frac{\partial P_j(\mathbf{x}_k, M, \mathbf{w})}{\partial u_{qr}} = \sum_{\ell=0}^{M} v_{j\ell} \frac{\partial z_\ell(\mathbf{x}_k, \mathbf{u})}{\partial u_{qr}} = v_{jq} \frac{\partial z_q(\mathbf{x}_k, \mathbf{u})}{\partial u_{qr}} = v_{jq} \frac{1}{\cosh^2\left(\sum_{s=0}^{n} u_{qs} x_{ks}\right)} x_{kr}$$

So, using the fact that $(1/\cosh^2 t) = 1 - \tanh^2 t$ and substituting these expressions back into Equations 6 and 4 gives:

$$v_{pq}^{new} = v_{pq}^{old} + 2\alpha \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \left[y_{kp} - P_p(\mathbf{x}_k, M, \mathbf{w}^{old}) \right] z_q(\mathbf{x}_k, \mathbf{u}^{old})$$

$$u_{qr}^{new} = u_{qr}^{old} + 2\alpha \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \left(\sum_{j=1}^{m} [y_{kj} - P_j(\mathbf{x}_k, M, \mathbf{w}^{old})] \, v_{jq}^{old} \right) [1 - z_q^2(\mathbf{x}_k, \mathbf{u}^{old})] \, x_{kr}$$

(note that the v_{jq}^{old} must be used). Now, if we interpret $\left[y_{kj} - P_j(\mathbf{x}_k, M, \mathbf{w}^{old}) \right]$ as an 'error' δ_{kj}, made by output neuron j on input-output example k (i.e., $(\mathbf{x}_k, \mathbf{y}_k)$), we can write:

$$v_{pq}^{new} = v_{pq}^{old} + 2\alpha \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \delta_{kp} \, z_q(\mathbf{x}_k, \mathbf{u}^{old}) \tag{9}$$

$$u_{qr}^{new} = u_{qr}^{old} + 2\alpha \lim_{N\to\infty} \frac{1}{N} \sum_{k=1}^{N} \left[\left(\sum_{j=1}^{m} \delta_{kj} \, v_{jq}^{old} \right) [1 - z_q^2(\mathbf{x}_k, \mathbf{u}^{old})] \, x_{kr} \right] \tag{10}$$

The first of these *learning laws* (Equation 9) was discovered (in the context of a perceptron with no hidden neurons where the output neurons directly receive **x** – a neural network architecture called an *ADALINE*) by Widrow and Hoff in 1959 [Widrow and Hoff, 1960] and is called the *delta rule* or LMS (Least Mean Squared) error rule. The second learning law (Equation 10) was discovered by Amari in

1966 [Amari, 1967], except for the formula in parentheses, namely $\left(\sum_{j=1}^{m} \delta_{kj} \, v_{jq}^{old} \right)$. This law is known as the *generalized delta rule* or *LMS rule*. It was the discovery of this formula in parentheses and, even more importantly, the realization of its vast implications, by Rumelhart, Hinton, and Williams in 1985 [Rumelhart, Hinton, and Williams, 1986; Rumelhart and McClelland, 1986] that triggered the 'neural network renaissance' which followed.

In Equation 9 the 'errors' δ_{kp} of the output neurons are used to calculate the output neuron weight changes. From Equation 10, these output-layer 'errors' (multiplied by the appropriate output neuron weights) must also be supplied to the hidden neurons in order to allow them to calculate their errors. This necessary 'backward transmission' of weighted errors from the output layer neurons back to the hidden layer neurons is known as *backpropagation of errors*. The learning process specified by Equations 9 and 10 is often referred to as *backpropagation learning*. Note that there must be one separate and distinct value backpropagated for each output layer non-bias weight.

Thus, given a large supply of examples $(\mathbf{x}_k, \mathbf{y}_k)$ chosen at random with respect to $\rho(\mathbf{x}, \mathbf{y})$, Equations 9 and 10 give us a means for incrementally adjusting $\mathbf{w}$ so as to converge to an optimal value $\mathbf{w}^*$ of $\mathbf{w}$. [Note: The practical process of selecting a suitable α will be discussed in the next section.] Although this *batch* backpropagation learning process will work, it is *very* slow (because, at each $\mathbf{w}^{old}$, a large number N of $(\mathbf{x}_k, \mathbf{y}_k)$ examples must be used to approximate the limits of Equations 9 and 10 needed to calculate $\mathbf{w}^{new}$).

Long ago (in the early 1950's, in another context [Kushner and Yin, 1997]), it was realized that batch learning is wasteful of computation. So, instead of averaging over a large number N of terms, as in Equations 9 and 10, why not change the weights with N = 1 examples and not average at all. This approach is called *jump-every-time learning*. Since it is obvious that not every weight adjustment of this type will lead to a lowering of F (as Theorem 3 guarantees will happen with the batch method), the immediate question becomes: Will jump-every-time backpropagation work? The answer is provided by

Theorem 4: Given a point $\mathbf{w}^{old}$ for which $\left| \nabla_{\mathbf{w}} F(M, \mathbf{w}) \, \right|_{\mathbf{w} = \mathbf{w}^{old}} \Big| > 0$, there exists a positive number α_0 such that for all α, $0 < \alpha \leq \alpha_0$,

$$E\left[F(M, \mathbf{w}^{new}) - F(M, \mathbf{w}^{old}) \right] < 0 , \tag{11}$$

where E[ ] is the expected value over all possible weight changes from $\mathbf{w}^{old}$ to $\mathbf{w}^{new}$, where $\mathbf{w}^{new}$ is derived from $\mathbf{w}^{old}$ in accordance with

$$v_{pq}^{new} = v_{pq}^{old} + 2 \, \alpha \, [\, y_{kp} - P_p(\mathbf{x}_k, M, \mathbf{w}^{old}) \,] \, z_q(\mathbf{x}_k, \mathbf{u}^{old}) \, ,$$

$$\text{for } p = 1, 2, \ldots, m \, ; \, q = 0, 1, 2, \ldots, M \tag{12}$$

$$u_{qr}^{new} = u_{qr}^{old} + 2 \, \alpha \left(\sum_{j=1}^{m} [y_{kj} - P_j(x_k, M, w^{old})] \, v_{jq}^{old} \right) [\, 1 - z_q^2(x_k, u^{old}) \,] \, x_{kr} \, ,$$

$$\text{for } q = 1, 2, \ldots, M \, ; \, r = 0, 1, 2, \ldots, n \tag{13}$$

determined by an ($\mathbf{x}_k$, $\mathbf{y}_k$) input-output example chosen at random with respect to $\rho(\mathbf{x}, \mathbf{y})$. For each admissible set a fixed α_0 value can be chosen that will work whenever both $\mathbf{w}^{old}$ and $\mathbf{w}^{old} + \theta\mathbf{h}$ belong to that set.

Proof: Letting $\mathbf{w}^{new} = \mathbf{w}^{old} + \mathbf{h}$, by Equation 2, we can write

$$F(M, \mathbf{w}^{new}) = F(M, \mathbf{w}^{old}) + \left[\nabla_{\mathbf{w}} F(M, \mathbf{w}) \big|_{\mathbf{w} = \mathbf{w}^{old}} \right] \cdot \mathbf{h} + \frac{1}{2} \mathbf{h}^T H(M, \mathbf{w}^{old} + \theta \mathbf{h}) \mathbf{h} \ ,$$

and since the expectation operator is linear, we can write

$$E\left[F(M, \mathbf{w}^{new}) - F(M, \mathbf{w}^{old}) \right] = \left[\nabla_{\mathbf{w}} F(M, \mathbf{w}) \big|_{\mathbf{w} = \mathbf{w}^{old}} \right] \cdot \ E[\mathbf{h}] + \frac{1}{2} E\left[\mathbf{h}^T H(M, \mathbf{w}^{old} + \theta\mathbf{h}) \mathbf{h} \right]$$

From Equations 9, 10, 12, and 13, we see that

$$E[\mathbf{h}] = -\alpha \nabla_{\mathbf{w}} F(M, \mathbf{w}) \ \big|_{\mathbf{w} = \mathbf{w}^{old}} .$$

Further, because of the boundedness of || H(M, **w**) || and the second terms of Equations 12 and 13, the term $\frac{1}{2} E[\mathbf{h}^T H(M, \mathbf{w}^{old} + \theta\mathbf{h})\mathbf{h}]$ will be a bounded factor times α^2. Thus, we get:

$$E\left[F(M, \mathbf{w}^{new}) - F(M, \mathbf{w}^{old}) \right] \leq -\alpha \left| \nabla_{\mathbf{w}} F(M, \mathbf{w}) \ \big|_{\mathbf{w} = \mathbf{w}^{old}} \right|^2 + \gamma \alpha^2$$

where γ is not dependent on α. So an α_0 sufficiently small will make the left side of this equation negative for all $0 < \alpha < \alpha_0$. Note that, since γ will be bounded on any admissible set, that we could also find a universal α_0 that would work everywhere where both $\mathbf{w}^{old}$ and $\mathbf{w}^{old} + \theta\ \mathbf{h}$ are in that set. ■

Finally, there are some additional facts concerning backpropagation learning which must be discussed. The most important of these is the existence of local extrema. In the discussion of learning presented above it was always assumed that $\nabla_{\mathbf{w}} F(M, \mathbf{w}) \neq \mathbf{0}$ and for admissible sets it was assumed that $\left| \nabla_{\mathbf{w}} F(M, \mathbf{w}) \big|_{\mathbf{w} = \mathbf{w}^{old}} \right| \geq c$, for some small positive constant c. This is fine, as long as the only places where the gradient goes to zero are global minima. However, it has been established for a closely related neural network architecture that the gradient can be the zero vector at places which are not global minima [Fukumizu and Amari, 2000]. While extrema which are not global minima could, in principle, be a problem for learning, experience has shown that the 'noisy' gradient descent offered by jump-every-time backpropagation does not get 'stuck' near such extrema. I suspect, but cannot yet prove, that all such non-global extrema are not local minima (i.e., they are 'saddle inflection points' like the curve $y = x^3$ at $x = 0$).

A real and serious problem that does often affect learning is *flat regions* on error surfaces. These are areas where the local surface slope is small in many directions. These regions are usually caused by having weight values which are too large; which cause the tanh functions to be operating out on their long flat tails. Being on a flat region slows down learning because, during each *learning cycle* (going

from $\mathbf{w}^{old}$ to $\mathbf{w}^{new}$), the gradient descent methods (both batch and jump-every-time) move the weight vector a distance which is roughly proportional to error surface slope (i.e., to the magnitude of the gradient). Further, the Hessian term can often be so large (relative to the gradient) that, given a value of α that worked fine in the admissible region learning started in, this term now interferes with the gradient term (decreasing the efficiency of stochastic gradient descent at points on this flat region – which is not part of the original admissible region) – further impeding learning progress.

Another problem is the frequent appearance of *troughs* in the error surface. These are essentially long, steep-sided valleys of shallow bottom slope. The learning process tends to spend most of its time climbing up and then down the valley walls; with only slow progress being made down the valley itself.

Although flat regions and troughs are problematic for learning, the solution is simple: either be patient and keep going or, if patience has given out, abandon this learning process and start a new one with new randomly chosen initial weights. Often, if sufficient patience is exercised, the learning process will move $\mathbf{w}$ on to a steeper part of the surface and faster progress towards a lower value of F(M,$\mathbf{w}$) will resume. If a new learning process is started, often its progress will be very dissimilar to the one that was abandoned and better downhill progress will occur.

In the next section the practical process of perceptron training and architecture adjustment (M control) is discussed.

7. The Perceptron Training and Architecture Adjustment Process

Perceptron training and architecture adjustment starts with a large set $\{(\mathbf{x}_k, \mathbf{y}_k)\}_{k=1}^{L}$ of randomly chosen input-output examples and ends up with a perceptron approximation $\mathbf{P}(\mathbf{x}, M, \mathbf{w})$ to the regression function $\mathbf{r}(\mathbf{x})$ of the probability density $\rho(\mathbf{x},\mathbf{y})$ underlying these examples. Between these endpoints lies a process which is described in this section (many variants of this process exist, but the version presented here has proven to be effective).

The first step in the process is selection of an initial M value. One approach to this is to let M equal the lesser of n and m, the number of inputs and outputs. Since refinement of the M value is the *perceptron architecture adjustment* part of the process, it doesn't matter much if this initial guess is wrong. Once this initial M has been selected it is fixed (until later in the process). The weights of the perceptron are then initialized to randomly selected small values, as discussed in Section 6.

The next step in the process is training of the perceptron $\mathbf{P}(\mathbf{x}, M, \mathbf{w})$ using randomly selected examples $(\mathbf{x}_k,\mathbf{y}_k)$ from the training set (refer to Section 2 for the definitions of the training set, training test set, verification set, and validation set).

For each training set example $(\mathbf{x}_k,\mathbf{y}_k)$ used, the current weight vector $\mathbf{w}^{old}$ of the perceptron is updated to the new weight vector $\mathbf{w}^{new}$ using Equations 12 and 13 (control of the learning rate α to be used is described below; an initial α value of 1 is often used). As training continues, the examples used are selected one at a time, in sequence, from the (randomly ordered) training set. If all examples have been used, then selection starts over at the first example of the training set and proceeds from there. Such a pass through the entire training set is known as a training *epoch*.

To evaluate training progress, the *learning curve* is usually displayed on the computer monitor. This is a plot of the value of F(M,$\mathbf{w}$) verses the number of training cycles, as shown in Figure 4. Usually, a point

is plotted on this graph at periodic intervals (e.g., every 1000 training cycles). The value of F(M,**w**) is

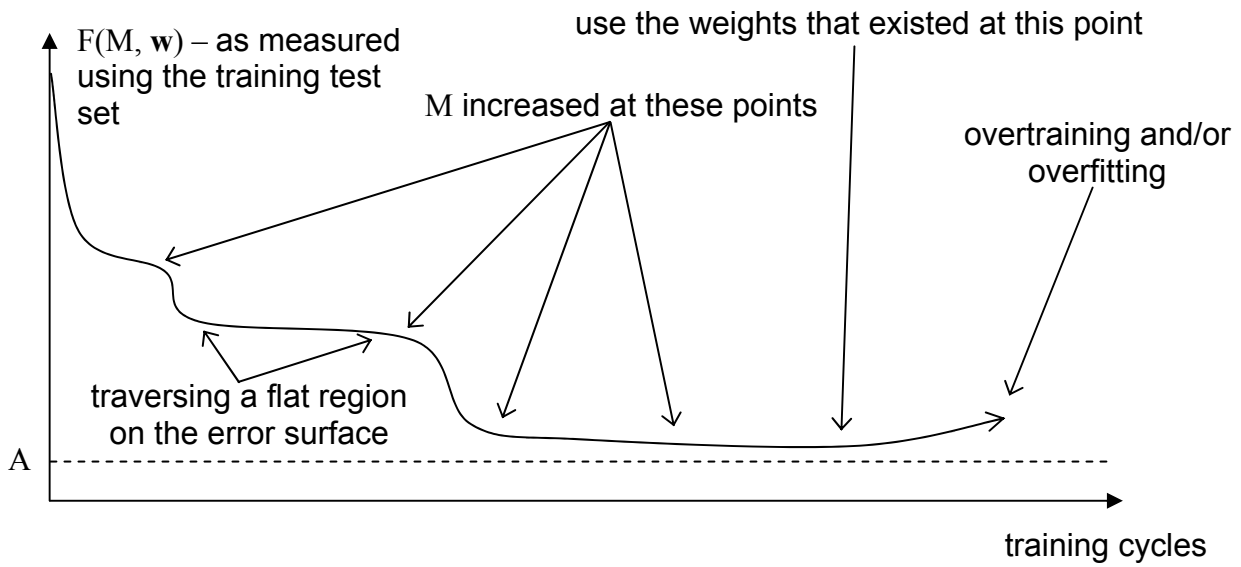

Figure 4. Perceptron learning curve. The learning curve is the primary tool used to gauge the progress of, and control, the perceptron training and architecture adjustment process.

approximated at each point to be plotted by freezing the current values of the perceptron weights and then using the S examples $(\mathbf{x}_k,\mathbf{y}_k)$ of the training test set to calculate

$$F(M,\mathbf{w}) \cong \frac{1}{S}\sum_{k=1}^{S} \left| \mathbf{y}_k - \mathbf{P}(\mathbf{x}_k, M, \mathbf{w}) \right|^2 .$$

As training progresses, one of the tasks is to refine the value of the learning rate α. This is done by increasing α tentatively until the learning curve starts showing occasional significant increases in F value. The value of α is then decreased until this behavior disappears. Adjusting α to the highest possible value will often significantly shorten overall training time. Sometimes it is necessary to adjust α several times as training progresses. It is critical that the weight vectors, at each of the points where F is calculated for the learning curve, be stored and carefully labeled. This way, if an α value which is too large is chosen, the training process can go back to the weight values that existed before the problem occurred. Also, storing the weight values at each of these points is important for architecture adjustment, as described below.

After training for a while, the learning curve will typically flatten out. This often indicates that training is beginning to reach its practical limit, or that a flat area has been encountered. It is a good idea to keep training for a little while in case it is a flat area. Then it is time to increase M. The increment of increase is often the initial M value, but it is really up to the practitioner. The initial weights of the new hidden neurons are again set to small random values, as are the output neuron weights which act upon the outputs of these new hidden neurons (the weights of the existing hidden neurons are retained as-is). Then training of all the weights continues. What often happens is that the value of F begins to significantly decrease again, as shown in Figure 4. Architecture adjustment (training to the point of leveling out followed by an increase in M) continues until further increases in M have no beneficial effect.

The learning curve can never go below the constant A, since this is the level of 'noise' intrinsic to the underlying density $\rho(\mathbf{x},\mathbf{y})$. Of course, we have no way of knowing the value of A. Thus, when the

learning curve flattens out and further increases in M have no beneficial effect, we can never be absolutely sure that we have reached the ultimate best performance level or whether we are simply on a flat region of the error surface. The remedy to this conundrum is to repeat the entire training and architecture adjustment process from start to finish several times using different small random initial weight values. If the same ultimate performance at the end of training is achieved (measured using the verification set once for each final trained perceptron), then this is probably near A. If a particular lowest F value is achieved on only a fraction of separate occasions, this is probably near A and the best of these networks should be used. In practice, as with non-global extrema and flat regions, *A ignorance* has not turned out to be a significant practical problem.

An additional technique which some practitioners add to the perceptron training methodology described here is *weight decay*. In weight decay, every weight in the perceptron is multiplied by a real constant slightly less than 1.0 (e.g., 0.9999) during every weight change (i.e., training) cycle. Weights whose values are frequently undergoing modification are largely unaffected by this. However, weights which might have been important earlier in training but which have not been modified for a very long time are gradually reduced in magnitude – until eventually they become close to zero. It is also typical that a test is periodically carried out to see if any of the hidden neurons have had all of their weights reduced to near-zero values; in which case those hidden neurons are removed – a process called *pruning*. One way to view weight decay and pruning is that under their influences the approximating surfaces of the $\mathbf{y}'$ components will have a tendency to revert to being 'flat' at zero elevation everywhere (as they are at the outset of training when the weights are all initialized to small values). In this view, the active influence of the training process keeps the 'important' weights 'pumped up' so that these surfaces take on the correct shape. Since weights sometimes do indeed become 'obsolete' during training, the weight decay and pruning techniques have proven to be useful practical adjuncts to perceptron training in some fields of application.

If training and architecture adjustment are continued beyond the point where the learning curve has almost certainly flattened out, a strange thing can happen: the learning curve can begin to ascend. This is due to *overtraining*. Overtraining is the result of excessive training on the training set (with its finite repertoire of examples) and testing on the training test set (which has different examples). What happens, in effect, is that the perceptron is being told that the training set contains the entire set of examples that exist (i.e., that the probability density $\rho(\mathbf{x},\mathbf{y})$ will essentially produce **only** these examples). Thus, the perceptron works hard to learn these examples exactly; at the expense of points lying between or beyond the training set examples. Performance on the training test set (which is the basis for calculating the learning curve) will typically slowly degrade as this specialization process continues. Overtraining is exacerbated by an M value which is larger than M_r. In this instance, the perceptron has more degrees of freedom than it needs to implement the regression function and it uses these to perform better on the training set examples – typically at the expense of performance on the training test set examples (this possible component of the overtraining problem is sometimes referred to as *overfitting*).

Overtraining can only occur when the same training data is used over and over again. A perceptron with M_r hidden neurons which is trained with randomly chosen $(\mathbf{x}_k,\mathbf{y}_k)$ examples, with no reuse of those examples (and with huge training test sets composed of still more not-reused examples), will (if the learning rate α is sufficiently slowly lowered to zero) converge to a perceptron equivalent to the regression perceptron $\mathbf{P}(\mathbf{x}, M_r, \mathbf{w}_r)$. The more training, the better the approximation of the regression perceptron. There is no probability of the learning curve turning upward because, in this no-data-reuse case, the jump-every-time backpropagation learning law continues to move downhill on the error surface and, in the limit, will converge to a global minimum. Even if the value of M being used is larger than

M_r , given all fresh data, the perceptron will still, in principle, eventually converge to one equivalent to $\mathbf{P}(\mathbf{x}, M_r, \mathbf{w}_r)$ (although weight decay and pruning can dramatically speed this up).

To avoid overtraining, all we need do is first decide that further M increases are not lowering the learning curve (or detect a possible distinctive upturn in the learning curve). We then go back to the point on the learning curve where the last increase in M did any good and take the final network to be this one at the point where M was increased again (i.e., at the point where it's performance became asymptotically constant and reached its best level). It is often useful to record the value of F at this point. This is an estimate of A. This final perceptron is taken as our approximation to the regression function.

In training a perceptron we are making the implicit assumption that our supply of data is sufficiently large. This assumption can be easily tested. All we need to do is temporarily discard the last half of the training set and the last half of the training test set. Then retrain the perceptron, starting with small random weights. If the performance of this new perceptron on the truncated training test set is comparable to that of the original perceptron on the original training test set, and if the performance of both perceptrons on the verification test set (which must, along with the validation test set, typically be much larger than the training test set) is similar, then we most likely have enough data. The perceptron trained on the truncated data sets must then be discarded. If the two neural networks have significantly different performance, then there is not enough data and use of the perceptron training and architecture adjustment process described here is not statistically valid.

It is useful to note that methods for manufacturing new data examples from existing examples have been developed and can sometimes be applied successfully when data is scarce. Typical of these is Abu-Mostafa's method of *hints* [Abu-Mostafa, 1995a and 1995b; Haykin, 1999]. In this method, a small random (e.g., Gaussian) vector perturbation $\mathbf{p}$ is added to the $\mathbf{x}$ vector of a training set example $(\mathbf{x}_k,\mathbf{y}_k)$, to form a new example $(\mathbf{x}_k + \mathbf{p}, \mathbf{y}_k)$. Several of these new *synthetic training examples* are typically created for each example in the training set – thus enlarging the training set considerably. This works very well if the magnitude of the covariance of the perturbation Gaussian is chosen judiciously and if the regression function being sought is very smooth and slowly varying. The hint-generated examples depend upon the assumption that as each $\mathbf{x}$ component is changed by a small amount the value of $\mathbf{r}(\mathbf{x})$ changes very little (in fact, not at all, in these new examples). When it is applicable and properly applied, the method of hints often yields good results.

If an adequate supply of data for building a perceptron cannot be obtained, it may still be useful to build a model – but one of a less capable type. The approach requiring the least data (but typically offering the lowest performance) is to build an *affine model* (see below). Another is to use one of the *ill-posed problem* methods (see Section 8).

An affine model can be easily constructed from the training set $\{(\mathbf{x}_k, \mathbf{y}_k)\}_{k=1}^{L}$ (assuming L is greater than both n and m) by letting $\mathbf{X}$ be the (n+1) x L matrix having column k with a 1 in the top row and with the components of $\mathbf{x}_k$ beneath it, in order; letting $\mathbf{Y}$ be the m x L matrix with column k being $\mathbf{y}_k$; and then letting $\mathbf{W}$ be the m x (n+1) matrix $\mathbf{W} = \mathbf{Y}\mathbf{X}^+$, where $\mathbf{X}^+$ is the *pseudoinverse* of $\mathbf{X}$ [Strang, 1988]. The *affine model* is then $\mathbf{y} = \mathbf{W}\mathbf{x}$, where $\mathbf{x}$ is an (n+1) x 1 vector with a 1 in the top row. [Technically, this is an *affine model* and not a *linear model* because bias values are used. An affine model usually has a lower mean squared error than a linear model.]

Even if the amount of data is very small (L only a small multiple of the largest of n and m) an affine model can sometimes be useful (although its meaning is rarely clear; unless the implicit assumption of

linear variable dependence somehow flows from the nature of the system being modeled). Often, an affine model can be simplified (to good advantage) by carrying out a *principal components analysis* on **W**. This is done via a standard linear algebra algorithm (which most numerical linear algebra algorithm libraries contain) called the *singular value decomposition* (SVD) [Strang, 1988] which expresses **W** as the product of three matrices: **W** = **UDV**; where **U** is an orthogonal m x m matrix, **D** is a diagonal m x (n+1) matrix with non-negative real values along its main diagonal in descending order, and **V** is an orthogonal (n+1) x (n+1) matrix. Typically, only a fraction of the values along the diagonal of **D** will be significantly above zero. If these larger values are left intact and all the other diagonal entries are set to zero, a new matrix $\check{\mathbf{D}}$ is created. We then let $\hat{\mathbf{W}}$ = **U**$\check{\mathbf{D}}$**V**. Often, the performance (in terms of mean squared error) of the model **y** = $\hat{\mathbf{W}}$**x** on new data will be superior to that of the model **y** = **Wx**. However, in most instances, an affine model exhibits overall poor performance (in comparison to a perceptron model for the same data environment) and is useful at all only if there is simply not enough data to build a more comprehensive model.

Once the final perceptron has emerged from training and architecture adjustment, it is time to test its performance using the verification set. Since the data in this set was not used in any way during training (unlike the training set, which was used to directly guide weight adjustment, and the training test set, which was used to guide architecture adjustment and training process stopping), this test is the most valid. Typically, the verification set is used only to test the relative performances of a few alternative perceptrons. The verification set performance typically gives a good estimate of the performance which can be anticipated on the validation set. The best performing perceptron is chosen and perceptron development is over.

The final step, once all perceptron development is over, is validation set testing. Because this set is used only once and because it is large enough to give a comprehensive test of performance, the results of this test are statistically valid and can be communicated to others. The other testing results do not have this character. Communication of these other results is not ethically acceptable; because they are not mathematically valid performance measurements.

Obviously, the process of perceptron training and architecture adjustment is computationally intensive. However, modern computers rarely have a problem carrying out this process (although in some cases it can take hours, or even days or weeks).

Perceptrons are now in use throughout the world in a wide variety of applications. This is a proven method that, in many cases, provides performance sufficiently close to optimal that users do not seek further improvement.

8. Discussion

The perceptron training and architecture adjustment process described in Section 7 represents a radical departure from classical viewpoints in statistical regression and pattern classification. The traditional view has been that the number of available examples for construction of a regression function is almost always too small to accurately define that function. This leads to the concept of an *ill-posed problem* (a problem for which the given data is insufficient to define a unique answer). From this mindset has flowed a vast and impressive literature of ideas and results about how to impose reasonable additional conditions so that the solution does become unique (or, at least practically so). In connection with regression and pattern classification this mindset has led to notions such as *regularization* (a crude implementation of Occam's razor: the smoothest and simplest candidate function which – in the

judgment of the investigator – 'reasonably fits' the known data is presumed to be the correct function) and *class separating feature function*s (selecting intermediate features which leave as much room as possible between points known to belong to different classes) [Cristianini and Shawe-Taylor, 2000; Vapnik, 2000].

The viewpoint adopted here is completely different from the ill-posed problem mindset. It is that, for many real-world problems, there is more than enough data to accurately determine the regression function. This is because, for reasons which are still not completely understood, perceptron convergence to the regression perceptron takes place with surprising rapidity. A hint of this can be seen in the Monte Carlo Theorem of Appendix B – where convergence in mean squared error takes place at a rate of $\frac{1}{N}$, independent of n and m (except to the extent that $\sigma^2(\mathbf{f})$ may indirectly depend on n and m). So, normally, all we have to do is carry out perceptron learning and we will make our way to the bottom of the error surface at a reasonable rate. The final result is an accurate approximation of the regression perceptron. If more data is available it won't do much good: after using the initial (already sufficiently large) supply of data we have a good approximation to the regression function and it will never change much, no matter how much additional data is supplied and how much additional training and architecture adjustment is carried out.

Of course, this *abundant data* mindset falls apart for problems where there isn't an adequate supply of data (as determined by the test described in Section 7). In this case, the perceptron is not applicable and alternatives must be considered. One alternative is the affine model described in Section 7. Another is to consult the literature on ill-posed problems (e.g., see [Cristianini and Shawe-Taylor, 2000] for an introduction)[6]. However, very often, today's regression problems (e.g., those in cognitive science) do not lack for data and the perceptron can be immediately and successfully applied.

The exposition of the perceptron presented here is only the cherry on the whipped cream on the top of the tip of an iceberg. A huge amount of knowledge about the perceptron and its myriad variants, and about methods for training and applying them, beyond that which can be presented here, has been accumulated[7]. This knowledge can be found in texts and monographs [Haykin, 1999; Fine, 1999; Vapnik, 2000], in relevant journals (e.g., *Neural Networks, Neural Computation, IEEE Transactions on Neural Networks, Neurocomputing,* etc.) and in conference proceedings (NIPS, ICNN, IJCNN, ECNN, ICONIP, etc.). Notwithstanding this large edifice of research, there remain many important unanswered questions about the perceptron.

Today's perceptron represents a monumental achievement of the human species. It is the product of millions of hours of work by thousands of researchers over more than four decades. Perceptron applications have produced thousands of new, permanent, high-quality jobs. Frank Rosenblatt, who began the investigation into perceptrons in 1956 (see Appendix A), would surely be pleased.

---

[6] Ill-posed problem methods still need large amounts of data to yield useful results. One way to view this is that ill-posed problem methods are valuable mostly when the amount of data available is only slightly smaller than that which would be required to build a perceptron. With less data than this, only something very simple, such as an affine model, should be attempted (and expectations must be kept low).

[7] For example, there is a *learnability theory* about which sorts of functions a particular class of function approximators (e.g., perceptrons), in the context of a particular measure of approximation accuracy (e.g., mean squared error), can learn to implement via a training process (see [Vapnik, 2000] for an introduction).

Appendix A: A Brief Perceptron History

In considering the history of the perceptron it is important to note that the formal perceptron definition has only recently begun to stabilize (in contrast to something like the definition of an abstract algebraic group; which has been stable for well over a century). Given this lack of definitional stability (even Rosenblatt himself changed his definitions multiple times), this Appendix will simply mention a number of main influences on the development process. This is not really satisfactory, but it is probably the best that can be done short of a book-length historical treatment (for a more extensive discussion see [Anderson and Rosenfeld, 1998]).

The idea that it might be possible to consider mathematical models of biological neurons goes back many decades [Anderson and Rosenfeld, 1988]. Before 1940, some attempts were made to mathematically model neural function, but none of these models were compelling. The first breakthrough came in 1943 with the seminal paper of McCulloch and Pitts [McCulloch and Pitts, 1943]. They postulated that neurons functioned as Boolean logic devices. While the logic circuit idea of McCulloch and Pitts was soon discredited as a biological theory, their work, and that of several others, led to a model of a neuron as a linear or affine sum followed by an *activation function* (usually a unit step function with a threshold offset). This neuron model (which, over time, began to be frequently, but incorrectly, attributed directly to McCulloch and Pitts) was widely adopted as a model for neuron behavior. Such mathematical neuron models are often termed *formal neurons*.

The thinking stimulated by McCulloch and Pitts and those they inspired didn't lead immediately to any important new concrete ideas, but it did generate a widespread feeling among prominent exponents of automated information processing (e.g., Norbert Weiner and John von Neumann, and many others) that building 'artificial brains' might become possible someday 'soon'. This excitement became even greater in 1949 with the publication of Hebb's hypothesis [Hebb, 1949] that neuronal learning involves experience-induced modification of synaptic strengths. Incorporating this hypothesis into the formal neuron (i.e., as variable weights) created many new avenues of possible investigation.

Although many investigations were launched in the early 1950s into neural networks composed of formal neurons with variable (*Hebbian*) weights; none of them yielded significant results until about 1957. At that point, two disparate themes emerged which, astoundingly, have still not been reconciled or connected: the *learnmatrix* and the *perceptron*. Studies of the learnmatrix associative memory neural network architecture were launched by Karl Steinbuch in about 1956 and led in 1961 to his writing of **Automat und Mensch** – the first technical monograph on artificial neural networks [Steinbuch, 1961; Steinbuch and Piske, 1963][8].

The perceptron (a term originally coined to mean a single threshold logic neuron designed to carry out a binary – i.e., two-class -- pattern recognition function) was developed by Frank Rosenblatt beginning in 1956. By 1957 he had developed a learning method for the perceptron and was soon able to prove mathematically that, in the case of linearly separable classes, the perceptron would be able to learn, by means of a Hebb-style supervised training procedure, to carry out its linear pattern classification function optimally. This was a major milestone in neural networks. For the first time, a formal neuron with trainable Hebbian weights was shown to be capable of carrying out a useful 'cognitive' function.

[8] Steinbuch's learnmatrix is a binary associative memory structure. His results were, and remain, impressive. However, since the connections between the learnmatrix and the perceptron have still not been established, Steinbuch's work, and its evolutes, will not be discussed further here.

Figure 5. Frank Rosenblatt, originator of the perceptron, c. 1959. Here, he was photographed, in connection with a television appearance, with the 'eye' (a 20 x 20 array of photoconductors) of the "Perceptron Mark I" pattern recognition neurocomputer. Note that the bundle of 400 wires in front of Rosenblatt, which make up the 'optic nerve' of the perceptron's visual system, are "randomly scrambled". The fact that the perceptron could adaptively learn to correct for such imprecise connections was one of its many appealing attributes. The widespread publicity of Rosenblatt's work, and that of his followers, stimulated (both in the press and among some researchers in the field) what might now be called an "irrational exuberance" regarding the prospects for rapid progress in the field of neural networks. This unwarranted exuberance, along with the widespread public attention, viscerally irritated and enraged a group of academics who (incorrectly, as it turned out) viewed themselves, and their ideas, as far more worthy of such optimism and attention. Before long, this group launched a vicious and brutal, but well-calculated and highly successful, attack against Rosenblatt in particular, and the field of neural networks in general. Rosenblatt's career declined and research on neural networks almost stopped for twenty years. Photo courtesy of Veridian Engineering.

Because of the limitations of digital computers in the 1950s, Rosenblatt was initially unable to try out his ideas on practical problems. So he and his colleagues successfully designed, built, and demonstrated a large analog circuit neurocomputer to enable experiments (see Figures 5, 6, 7, 8, 9, and 10). His machine (called the Perceptron Mark I) had 512 adjustable weights and a crude 400-pixel camera as its

visual sensor. It successfully demonstrated an ability to recognize alphabetic characters. Rosenblatt's work (which was widely publicized, including numerous popular magazine articles and appearances on major network television programs) inspired a large number of investigators to begin work on neural networks.

Figure 6. The Perceptron Mark I neurocomputer at the Cornell Aeronautical Laboratory c.1958 with Charles Wightman, the engineer who designed and constructed it. The Perceptron's 'eye' is housed in the camera body seen in the foreground. The perceptron itself consists of the racks of equipment in the background. The photoconductors of the 'eye' were so insensitive that for them to respond to an alphabetic character (such as the C shown here), the character had to have high contrast and be illuminated at point-blank range by the four powerful spotlights shown. When in full operation, the Mark I must have consumed a considerable amount of power. One can imagine the lights of Ithaca dimming while the perceptron learned to recognize the letters of the alphabet. Photo courtesy of Veridian Engineering.

By 1962, Rosenblatt's book (the second monograph on neural networks) **Principles of Neurodynamics** [Rosenblatt, 1962] discussed a more advanced sort of perceptron – one similar to that discussed in this book. However, a big unsolved problem remained: an effective method for training the hidden layer neurons.

Another important development of the late 1950s was the development of the *ADALINE* (ADAptive LINear NEuron – but it was actually affine!) by Widrow and Hoff [Widrow and Hoff, 1960]. The learning law for this network was the delta rule used in the output layer neurons of the perceptron discussed in this book. Widrow's perspicuous mathematical derivation of the delta rule learning law set the foundation for many important later developments. As with Rosenblatt, Widrow turned to analog electronics to build an implementation of the ADALINE (including development of a variable-electrical-resistance weight implementation device called a MEMISTOR [Hecht-Nielsen, 1990]).

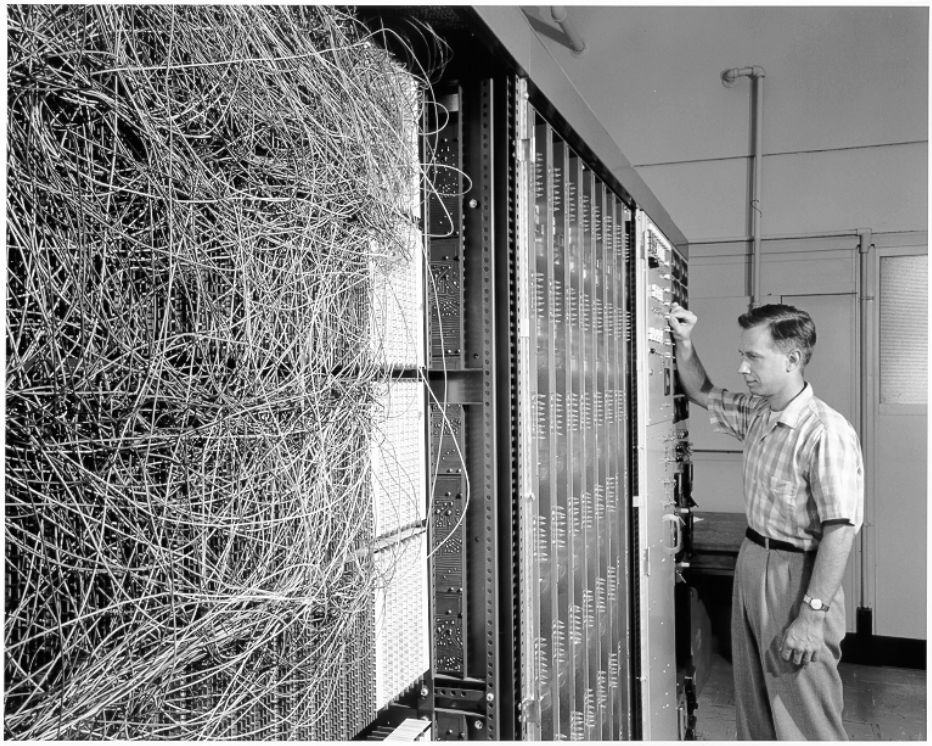

Figure 7. Charles Wightman running the Perceptron Mark I. The 'rats nest' of wires in the foreground are at the patch panels where the electrical connections of the neural network were made. These connections were typically set by using a book of random numbers. That a randomly wired perceptron could learn to correctly recognize characters was interpreted by many in the late 1950's as strong evidence that the perceptron was just a step away from a brain. The rack between the patch panels and the control panel (the one in the background where Wightman is making an adjustment) contained the adaptive elements of the perceptron. Photo courtesy of Veridian Engineering.

A development which was understood only by a few experts at the time, but which is now recognized as a phenomenally prescient insight, was the discovery in 1966 of the essential part of the generalized delta rule by Amari [Amari, 1967]. The only thing missing was how to calculate the hidden neuron errors.

By the mid 1960s many of the researchers who had been attracted to the field in the late 1950s began to become discouraged. No significant new ideas had emerged for some time and there was no indication that any would for a while. Many people gave up and left the field. Contemporaneously, a series of coordinated attacks were lodged against the field in talks at research sponsor headquarters, talks at technical conferences and talks at colloquia. These attacks took the form of rigorous mathematical arguments showing that the original perceptron and some of its relatives were mathematically incapable of carrying out some elementary information processing operations (e.g., the Exclusive-OR logical operation). By considering several perceptron variants and showing all of them to be 'inadequate' in this same way, an implication was conveyed that all neural networks were subject to these severe limitations. These arguments were refined and eventually published as a book [Minsky and Papert, 1969] (see [Anderson and Rosenfeld, 1998] for a more thorough discussion). The final result was the emergence of a pervasive conventional wisdom (which persisted until 1985) that 'neural networks have been mathematically proven to be useless'.

Although the missing piece of the generalized delta rule (i.e., backpropagation) was independently discovered by at least two people during the period 1970-1985 [Anderson and Rosenfeld, 1998], namely Paul Werbos in 1974 and David Parker in 1982, these discoveries had little impact and did not become

widely known until after the work of Rumelhart, Hinton, and Williams in 1985 [Rumelhart et al, 1986]. As can be seen by referring to any current text on neural networks (e.g., [Haykin, 1999; Fine, 1999]), enormous progress has occurred since then. The faith that Rosenblatt (who died in a boating accident in 1971) had in the perceptron has been resoundingly vindicated.

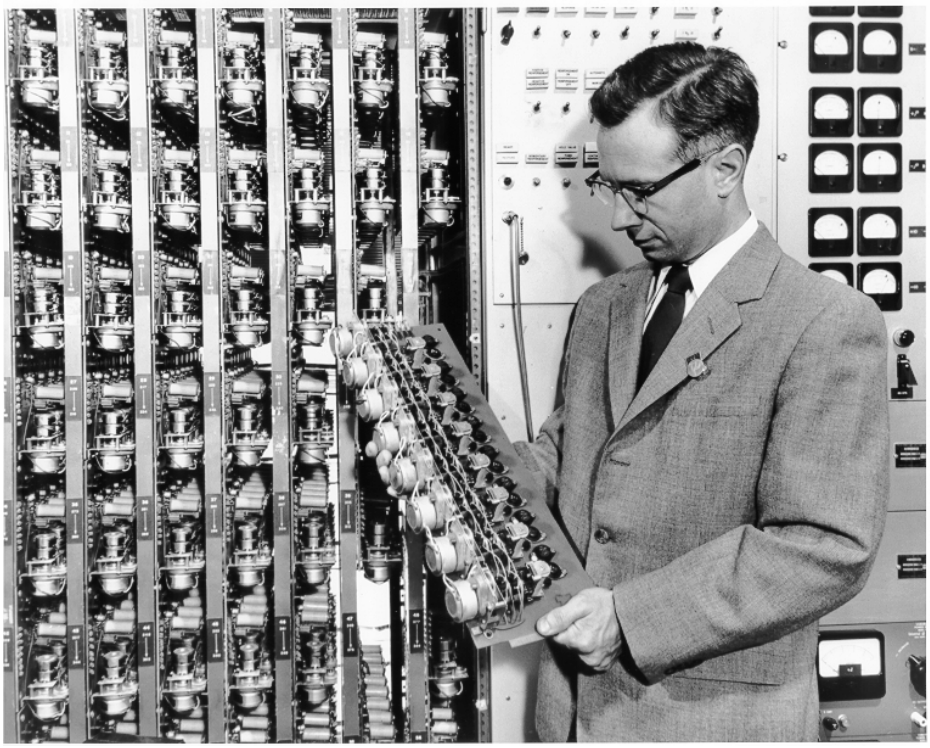

Figure 8. Charles Wightman holding a subrack of eight Perceptron adaptive weight implementation units. There were a total of eight rows and eight columns of such subracks – for a total of 512 weight implementation units. Each weight implementation unit consisted of a geared electric motor driving the shaft of a rotary potentiometer (a scheme which had been pioneered by Marvin Minsky several years earlier in his *Snark* neurocomputer [Anderson and Rosenfeld, 1998][9]). The potentiometer resistance R was the inverse of its weight value W (which had to be a positive number in a relatively small range). By Ohm's law, V = IR, the voltage drop V across the potentiometer equals the current running through it multiplied by the potentiometer's resistance. Rewriting this equation gives I = W V (where, again, W = 1 / R). The input voltage V is, for example, the output voltage of one of the eye's photoconductors. By simply connecting together the outputs of all of the weighting units of one perceptron and then grounding this lead, the current in this summed output (which, by Kirchhoff's law, is the sum of the individual weighting unit currents) will be equal to $\sum_{i=1}^{n} W_i V_i$. Thus, the Minsky design is a way of building an analog electric weighted sum with adjustable weights. Photo courtesy of Veridian Engineering.

[9] Minsky was a key leader of the attack against Rosenblatt and the field of neural networks. Rosenblatt's great success with Minsky's hardware scheme (the history of which the press failed to mention), as well as issues surrounding the fact that Minsky and Rosenblatt had attended the same high school, probably added additional fuel to the fire.

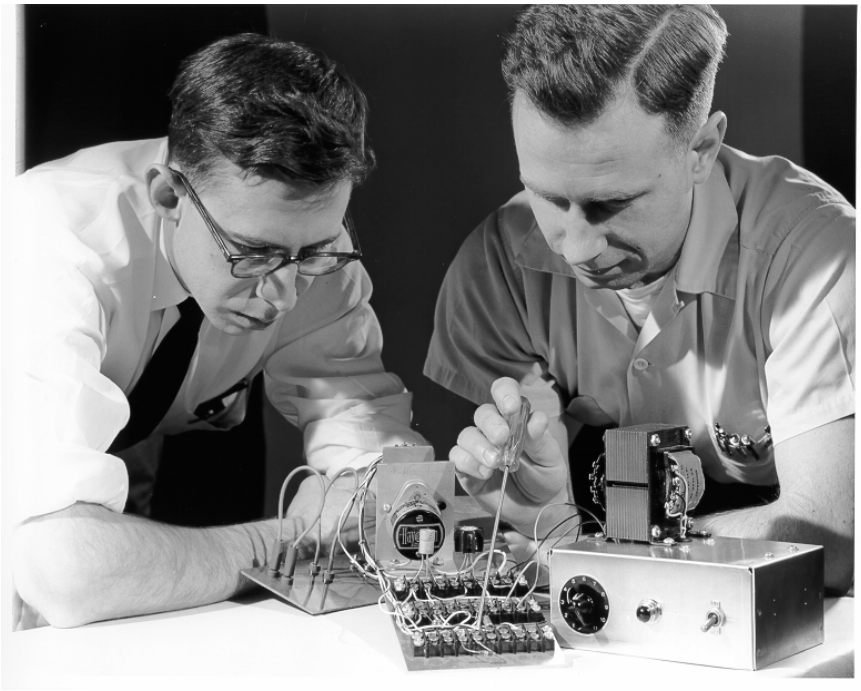

Figure 9. Rosenblatt and Wightman working on a Perceptron component. Photo courtesy of Veridian Engineering.

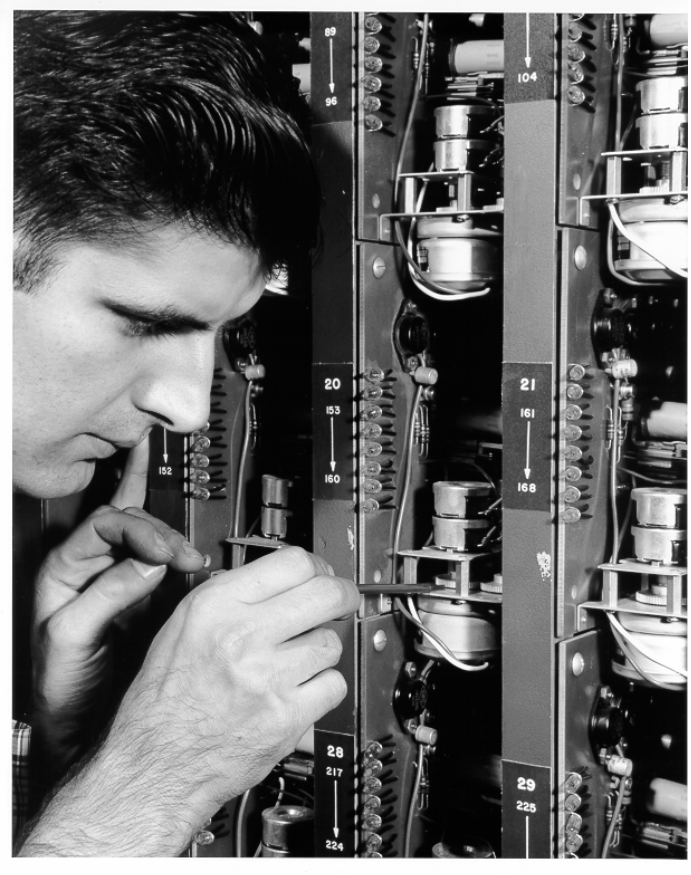

Figure 10. A close-up showing some weight implementation units of the Perceptron Mark I. The lights were used to monitor the learning process. Today's computer implementations of perceptrons provide a performance improvement of over nine orders of magnitude, compared to the Mark I, in both processing speed and physical volume. Photo courtesy of Veridian Engineering.

Appendix B: The Monte Carlo Theorem

Let:

- $\rho(\mathbf{x})$ be a bounded continuous probability density of compact support on R^n.
- $\mathbf{f}(\mathbf{x})$ be a bounded continuous function of compact support on R^n.
- $E[\mathbf{f}] \equiv \int_{R^n} \mathbf{f}(\mathbf{x})\,\rho(\mathbf{x})\,d\mathbf{x}$ and $\sigma^2(\mathbf{f}) \equiv \int_{R^n} |\mathbf{f}(\mathbf{x}) - E[\mathbf{f}]|^2\,\rho(\mathbf{x})\,d\mathbf{x}$. Note: $\int_{R^n} \rho(\mathbf{x})\,d\mathbf{x} = 1$
- $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$ be a sequence of N vectors in R^n chosen independently at random with respect to $\rho(\mathbf{x})$.

Theorem [Monte Carlo Theorem (von Neumann & Ulam c. 1950)]:

For each integer $N, N > 2$:

$$\underset{\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}}{E}\left[\left|\frac{1}{N}\sum_{i=1}^{N}\mathbf{f}(\mathbf{x}_i) - \int_{R^n}\mathbf{f}(\mathbf{x})\;\rho(\mathbf{x})\,d\mathbf{x}\right|^2\right] = \frac{\sigma^2(\mathbf{f})}{N}$$

Proof:

$$E\left[\left|\frac{1}{N}\sum_{i=1}^{N}\mathbf{f}(\mathbf{x}_i) - \int_{R^n}\mathbf{f}(\mathbf{x})\,\rho(\mathbf{x})\,d\mathbf{x}\right|^2\right] \equiv$$

$$\int_{R^n}\int_{R^n}\cdots\int_{R^n}\left|\frac{1}{N}\sum_{i=1}^{N}\mathbf{f}(\mathbf{x}_i) - \int_{R^n}\mathbf{f}(\mathbf{x})\,\rho(\mathbf{x})\,d\mathbf{x}\right|^2 \rho(\mathbf{x}_1)\,d\mathbf{x}_1\;\rho(\mathbf{x}_2)\,d\mathbf{x}_2\cdots\rho(\mathbf{x}_N)\,d\mathbf{x}_N$$

Now let $g_j(\mathbf{x}_k) = f_j(\mathbf{x}_k) - \int_{R^n} f_j(\mathbf{x})\,\rho(\mathbf{x})\,d\mathbf{x} = f_j(\mathbf{x}_k) - E[\,f_j\,]$, where the subscript j means the j^{th} component of the respective vector function. Then, using the fact that $\mathbf{a} = \sum_{j=1}^{n}[\,a_j\,]^2$ we can write

$$E\left[\left|\frac{1}{N}\sum_{i=1}^{N}\mathbf{f}(\mathbf{x}_i) - \int_{R^n}\mathbf{f}(\mathbf{x})\,\rho(\mathbf{x})\,d\mathbf{x}\right|^2\right] \equiv \int_{R^n}\int_{R^n}\cdots\int_{R^n}\left|\frac{1}{N}\sum_{i=1}^{N}\mathbf{f}(\mathbf{x}_i) - \int_{R^n}\mathbf{f}(\mathbf{x})\,\rho(\mathbf{x})\,d\mathbf{x}\right|^2 \rho(\mathbf{x}_1)\,d\mathbf{x}_1\;\rho(\mathbf{x}_2)\,d\mathbf{x}_2\cdots\rho(\mathbf{x}_N)\,d\mathbf{x}_N$$

$$= \sum_{j=1}^{n}\int_{R^n}\int_{R^n}\cdots\int_{R^n}\left[\frac{1}{N}\sum_{i=1}^{N}f_j(\mathbf{x}_i) - \int_{R^n}f_j(\mathbf{x})\,\rho(\mathbf{x})\,d\mathbf{x}\right]^2 \rho(\mathbf{x}_1)\,d\mathbf{x}_1\;\rho(\mathbf{x}_2)\,d\mathbf{x}_2\cdots\rho(\mathbf{x}_N)\,d\mathbf{x}_N$$

$$= \sum_{j=1}^{n}\int_{R^n}\int_{R^n}\cdots\int_{R^n}\left[\frac{1}{N}\sum_{i=1}^{N}g_j(\mathbf{x}_i)\right]^2 \rho(\mathbf{x}_1)\,d\mathbf{x}_1\;\rho(\mathbf{x}_2)\,d\mathbf{x}_2\cdots\rho(\mathbf{x}_N)\,d\mathbf{x}_N$$

$$= \frac{1}{N}\sum_{j=1}^{n}\sum_{i=1}^{N}\int_{R^n}\int_{R^n}\cdots\int_{R^n} g_j^2(\mathbf{x}_i)\ \rho(\mathbf{x}_1)\,d\mathbf{x}_1\ \rho(\mathbf{x}_2)\,d\mathbf{x}_2\cdots\rho(\mathbf{x}_N)\,d\mathbf{x}_N$$

$$+ \frac{2}{N}\sum_{j=1}^{n}\sum_{1\le i<k\le N}\int_{R^n}\int_{R^n}\cdots\int_{R^n} g_j(\mathbf{x}_i)\,g_j(\mathbf{x}_k)\ \rho(\mathbf{x}_1)\,d\mathbf{x}_1\ \rho(\mathbf{x}_2)\,d\mathbf{x}_2\cdots\rho(\mathbf{x}_N)\,d\mathbf{x}_N$$

$$= \frac{1}{N}\sum_{j=1}^{n}\int_{R^n} g_j^2(\mathbf{x})\ \rho(\mathbf{x})\,d\mathbf{x}$$

$$+ \frac{2}{N}\sum_{j=1}^{n}\sum_{1\le i<k\le N}\int_{R^n} g_j(\mathbf{x}_i)\left[\int_{R^n} g_j(\mathbf{x}_k)\ \rho(\mathbf{x}_k)\,d\mathbf{x}_k\right]\rho(\mathbf{x}_j)\,d\mathbf{x}_j$$

However,

$$\int_{R^n} g_j(\mathbf{x}_k)\ \rho(\mathbf{x}_k)\,d\mathbf{x}_k = \int_{R^n}\left[f_j(\mathbf{x}_k) - E\left[f_j\right]\right]\rho(\mathbf{x}_k)\,d\mathbf{x}_k$$
$$= \int_{R^n} f_j(\mathbf{x}_k)\ \rho(\mathbf{x}_k)\,d\mathbf{x}_k - E\left[f_j\right]\int_{R^n}\rho(\mathbf{x}_k)\,d\mathbf{x}_k = E\left[f_j\right] - E\left[f_j\right] = 0$$

and so we get

$$E\left[\left|\frac{1}{N}\sum_{i=1}^{N}\mathbf{f}(\mathbf{x}_i) - \int_{R^n}\mathbf{f}(\mathbf{x})\,\rho(\mathbf{x})\,d\mathbf{x}\right|^2\right] = \frac{1}{N}\sum_{j=1}^{n}\int_{R^n} g_j^2(\mathbf{x})\ \rho(\mathbf{x})\,d\mathbf{x}$$

$$= \frac{1}{N}\int_{R^n}\left|\mathbf{g}(\mathbf{x})\right|^2\rho(\mathbf{x})\,d\mathbf{x}$$
$$= \frac{1}{N}\int_{R^n}\left|\mathbf{f}(\mathbf{x}) - E\left[\mathbf{f}\right]\right|^2\rho(\mathbf{x})\,d\mathbf{x} \equiv \frac{\sigma^2(\mathbf{f})}{N}$$

because this last integral is the definition of the variance $\sigma^2(\mathbf{f})$ of $\mathbf{f}$. ∎

Appendix C: Radial Basis Function Perceptrons

A *radial basis function* $\boldsymbol{\psi}$ [Poggio and Girosi 1990 and 1993] is a function

$$\boldsymbol{\psi} : R^n \rightarrow R^m$$
$$\mathbf{x} \mapsto \mathbf{y} = \boldsymbol{\psi}(\mathbf{x})$$

of the form

$$\boldsymbol{\psi}(\mathbf{x}) = \sum_{i=1}^{M} \mathbf{c}_i \, \theta(|\, \mathbf{x} - \mathbf{d}_i \,|^2)$$

where the $\mathbf{c}_i$ are constant vectors in R^m, the $\mathbf{d}_i$ are constant vectors in R^n, the real-valued function θ is zero outside some interval $[0,\lambda]$, and $\theta(0) = 1.0$. Clearly, if the points $\mathbf{d}_i$ are all spaced at distances greater than $\sqrt{\lambda}$ from each other, then we will have $\boldsymbol{\psi}(\mathbf{d}_i) = \mathbf{c}_i$, for i = 1, 2, …, M . Usually, the function θ is chosen to be monotonically decreasing so that if multiple of the 'squared-radius' intervals $[0,\lambda]$ around different $\mathbf{d}_i$'s overlap, the value of $\boldsymbol{\psi}$ will be a 'blending' of their $\mathbf{c}_i$ vectors (with the closest $\mathbf{d}_i$'s supplying the largest contributions to this weighted sum).

Note that if we somehow select and fix the M value and the $\mathbf{d}_i$ vectors, we can use an example data set $\{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_L, \mathbf{y}_L)\}$ to find a set of $\mathbf{c}_i$ vectors by solving the linear equation $\mathbf{CD} = \mathbf{Y}$ for $\mathbf{C}$ (where $\mathbf{C}$ is the (n x M) matrix whose columns are the $\mathbf{c}_i$ vectors, $\mathbf{D}$ is the (M x L) matrix whose columns are the

$$\left(\theta(|\, \mathbf{x}_k - \mathbf{d}_1 \,|^2), \theta(|\, \mathbf{x}_k - \mathbf{d}_2 \,|^2), \ldots, \theta(|\, \mathbf{x}_k - \mathbf{d}_M \,|^2)\right)^T$$

vectors, and $\mathbf{Y}$ is the (m x L) matrix whose columns are the $\mathbf{y}_k$ vectors. From basic linear algebra [Strang, 1988], the minimum mean squared error solution to this linear equation having minimum norm is $\mathbf{C} = \mathbf{YD}^+$, where $\mathbf{D}^+$ is the pseudoinverse of $\mathbf{D}$.

One strong appeal of radial basis functions is this ability to carry out the 'training' process by simply calculating $\mathbf{YD}^+$ (which can usually be accomplished using canned linear algebra software). The problem is choosing M and the $\mathbf{d}_i$ vectors. Many methods for doing this have been developed, but this topic will not be discussed further here (see [Haykin, 1999] for further information).

Intuitively, the radial basis function seems like it might be able to 'interpolate' its output for $\mathbf{x}$ vectors lying between the $\mathbf{d}_i$ vectors. Unfortunately, fundamental geometrical facts about high-dimensional spaces (principally and astoundingly, that the distance between any two $\mathbf{d}_i$ vectors, as they are typically chosen, will be approximately the same) confound this intuition (i.e., each x lies roughly equidistant from all the $\mathbf{d}_i$'s). Nonetheless, the radial basis function concept still has merit in connection with perceptrons.

An important observation, which was made by Casasent [Casasent and Barnard, 1990; Telfer and Casasent, 1991; Denoeux and Lengelle, 1993] and by Poggio [Poggio and Girosi 1990 and 1993], is that perceptrons can implement a radial basis function form as an initial starting state, and then go on to be trained in the usual way. This procedure has proven to be of value in several practical fields of application.

The key insight is that we can write $\alpha \, | \mathbf{x} - \mathbf{d}_i |^2$ (where α is a positive constant) in the following form

$$\alpha \, | \mathbf{x} - \mathbf{d}_i |^2 = \alpha \, | \mathbf{x} |^2 - 2 \, \alpha \, \mathbf{x} \bullet \mathbf{d}_i + \alpha \, | \mathbf{d}_i |^2$$

and that the weighted sum of a hidden-layer neuron of a perceptron can implement this form; as long as we supply $|\mathbf{x}|^2$ as an additional input to the perceptron. The weight on this new $|\mathbf{x}|^2$ input must clearly be α; the weight on the j^{th} component of $\mathbf{x}$ ($j = 1, 2, \ldots, n$) must clearly be $-2 \, \alpha \, d_{ij}$, and the bias weight must clearly be $\alpha \, |\mathbf{d}_i|^2$.

If we now construct a perceptron with such a hidden neuron, the output of this neuron will be

$$\tanh \left(\alpha \, | \mathbf{x} - \mathbf{d}_i |^2 \right)$$

which will rise from 0.0 when $\mathbf{x} = \mathbf{d}_i$ to 1.0 as $\mathbf{x}$ moves away from $\mathbf{d}_i$. Casasent noted that if we subtract this hidden neuron output from 1.0, that this difference behaves roughly as a term $\theta(| \mathbf{x} - \mathbf{d}_i |^2)$ in a radial basis function.

By adding $c_{\ell i}$ to the bias weight of output neuron ℓ ($\ell = 1, 2, \ldots, m$), which is initially zero, and setting the weight of this neuron acting upon the input from hidden neuron i to the value $-c_{\ell i}$, the vector output of the perceptron's output layer neurons due to the input from hidden neuron i will be

$$\mathbf{c}_i \left[1 - \tanh \left(\alpha \, | \mathbf{x} - \mathbf{d}_i |^2 \right) \right]$$

and the total vector output of the perceptron due to all of its hidden neurons will be

$$\sum_{i=1}^{M} \mathbf{c}_i \left[1 - \tanh \left(\alpha \, | \mathbf{x} - \mathbf{d}_i |^2 \right) \right] \quad . \qquad \text{(A1)}$$

Clearly, this perceptron will implement a structure which is essentially a radial basis function with a function $\theta(s)$ given by $1 - \tanh(s)$ (with s restricted to being positive). By choosing the value of α properly, the extinction point λ can be controlled (actually, there is no reason why the value of α could not be chosen differently for each hidden neuron). If a function θ with a different form is desired, then multiple hidden units, having different α values and utilizing slightly offset bias values (to 'shift' each one over in radius squared) can be used with the appropriately scaled values of the output unit weights used above. However, experience has shown that the form A1 is almost always satisfactory.

Given such a *radial basis function perceptron*, how do we choose the $\mathbf{c}_i$ and the $\mathbf{d}_i$? One very appealing way is to *load in* M randomly chosen $(\mathbf{x}_k, \mathbf{y}_k)$ training set examples $\{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_M, \mathbf{y}_M)\}$. This is accomplished by letting $\mathbf{c}_i = \mathbf{y}_i$ and $\mathbf{d}_i = \mathbf{x}_i$. Assuming that the α value(s) are chosen so that the hidden layer 'radial basis functions' do not overlap significantly, this perceptron will have the property that

$$\mathbf{P}(\mathbf{x}_k, M, \mathbf{w}) = \mathbf{y}_k$$

for k = 1, 2, …, M. In other words, it will be 'born trained' for the examples used in its construction. No linear algebra manipulations are required. [Of course, in general, the $\mathbf{y}_k$ outputs are noisy and do not necessarily accurately represent the outputs of the regression function.]

After a collection of examples $\{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_M, \mathbf{y}_M)\}$ is loaded into a radial basis function perceptron, the perceptron can then be trained further using the standard perceptron training procedure. Compared to the usual procedure of starting all of the weights out at small values, this neural network begins its training already functioning 'correctly' for a large number of examples. Typically, the M examples used to initialize the radial basis function perceptron can be reused in the training set without harm. When used with weight decay and pruning (which can take an initially large hidden layer and reduce its size), this technique can yield a suitable final perceptron, sometimes with considerably less overall computational effort than the standard technique.

For some applications, the fact that the initial perceptron gives **y** example outputs at each of the trained **x** values (and – if the regression function being approximated is smooth and slowly-varying and if the α constant(s) are chosen appropriately – at nearby values) can be exploited by not even training the radial basis function perceptron at all. In other words, for certain problems, the initial loaded perceptron can function (at least temporarily) as an approximate implementation of the regression function.

A neural network similar to a radial basis function perceptron, is the *support vector machine* [Vapnik, 2000]. Training of this network is accomplished via a quadratic optimization algorithm. However, support vector machines typically have huge numbers of hidden neurons (which makes their implementation in real-time applications slow) and they are computationally difficult to train using large training sets with large n. Further, the size of the hidden layer, and therefore, the number of computations required for training and implementation, scale roughly linearly with m (the dimension of the **y** vectors); unlike the perceptron, where the hidden neurons are often 'shared' by all of the output neurons. Support vector machines are primarily used for two-class pattern classification (see Appendix E). Classification with more than two classes is usually done by building a separate classifier for each pair of classes.

The required amount of training and testing data needed for support vector machine construction is typically much larger than that needed for training and testing a perceptron of equivalent final performance. Support vector machines for regression have been defined; but seem to offer little in comparison to the perceptron.

The rather complicated methodology for building a support vector machine has been mechanized in a number of well-designed, highly automated, readily available, high-quality software packages. When appropriately applied, these packages yield good models with minimal user effort and knowledge. However, many examples of mis-application of such tools exist. The usual problem is an insufficient quantity of training and testing data.

Appendix D: Data Dimensionality Reduction

For reasons that remain somewhat mysterious, real-world systems often produce behavioral description (i.e., 'input-output') $\{(\mathbf{x}_k, \mathbf{y}_k)\}_{k=1}^{L}$ data vectors that are inherently highly structured (the mystery probably lies more in our inadequate understanding of high-dimensional geometry than in the reality that 'many real-world systems have intrinsic simplicity due to the constrained relationships between their parts'). By "highly structured" it is meant that the data vectors (**x**'s or **y**'s) lie almost completely within an affine *data subspace* of R^n or R^m having an astoundingly smaller dimension than n or m. [Note: it is assumed here that both n and m are large (say, greater than 10). If either n or m is not large, then it is usually better to simply leave those data vectors alone and only apply the usual prescaling and z-scaling to them.]

The main point of this Appendix is that a modified version of the affine model generation process discussed in Section 7 can often be used to find this data subspace (this process is done separately for the **x** vectors and for the **y** vectors). Once this much-lower-dimensional data subspace has been found, the relevant vectors can be re-expressed as sums of basis vectors which lie in it and then perceptron development can proceed using a much smaller number of input and/or output variables. This is advantageous, since the perceptron will therefore have many fewer weights and will presumably be easier to train.

Besides providing a beneficial problem simplification, this *data subspace determination* procedure can be used to crudely analyze the internal structure of the data by developing *eigenfeatures* (of quantifiable importance) that form a basis set for the data vectors. To simplify the notation, only the analysis of **x** vectors will be discussed here. The same analysis can be carried out separately for the **y** vectors.

The first step in data subspace determination is to (using the procedure described in Section 2) prescale and then z-scale the entire body $\{\mathbf{x}_k\}_{k=1}^{L}$ of **x** data vectors (it is assumed that erroneous outliers have been removed before z-scaling, so that the sample mean of the entire data set is the zero vector). These preprocessing steps are essential and mandatory (the data vectors must have zero mean and the scalings of the **x** components must be comparable).

Next, we take the matrix **X** to be the n x L matrix having its k^{th} column equal to $\mathbf{x}_k$, where k = 1, 2, …, L (note that this is different than for building an affine model – bias values are not needed here because the **x** sample mean is the zero vector and because we are not attempting to build an input-output model). Then we let **C** be the n x n matrix $\mathbf{C} \equiv \mathbf{X}\mathbf{X}^{+}$, where, as in Section 7, $\mathbf{X}^{+}$ is the pseudoinverse of **X** [Strang, 1988].

One of the defining properties of the pseudoinverse is that $(\mathbf{X}\mathbf{X}^{+})\mathbf{X} = \mathbf{X}$ [Strang, 1988]. Because the columns of **X** are the $\mathbf{x}_k$, this relationship implies that

$$\mathbf{C}\mathbf{x}_k \equiv (\mathbf{X}\mathbf{X}^{+})\mathbf{x}_k = \mathbf{x}_k \quad \text{for } k = 1, 2, \ldots, L$$

Thus, if we define the *mean squared error (MSE) of data representation* G as

$$G \equiv \frac{1}{L} \sum_{k=1}^{L} |\mathbf{x}_k - \mathbf{C}\mathbf{x}_k|^2 \quad ,$$

we see that G = 0. In other words, the matrix **C** is constructed in such a way that it's action naturally preserves all of the $\mathbf{x}_k$ data vector examples from which **X** is constructed. If the L $\mathbf{x}_k$ data vectors have been chosen at random with respect to some probability density, and if L is large enough, then C will typically have almost exactly the same effect on new randomly chosen **x** examples (i.e., **Cx** will be very close to **x**). The amazing and mysterious part of this story is that, notwithstanding its lack of effect on the $\mathbf{x}_k$ data vectors, for many real-world data sets, the matrix **C** is very far from being the n x n identity matrix **I**.

Since **C** is symmetric (because another defining property of the pseudoinverse is that $(\mathbf{XX}^+)^T = \mathbf{XX}^+$ [Strang, 1988]), it is easy to show [Strang, 1988] that a singular valued decomposition (*SVD* – see Section 7) of **C** (which is not always unique) will have the form $\mathbf{C} = \mathbf{UDU}^T$; where **U** is an orthogonal n x n matrix with pairwise-orthogonal, unit-length, vector columns $\mathbf{u}_i$, i = 1, 2, …, n , and **D** is a diagonal n x n matrix with non-negative real values d_i , i = 1, 2, …, n along its main diagonal, ordered in descending magnitude (i.e, $d_i \geq d_{i+1} \geq 0$ for all i). Because **U** is orthogonal, the inverse of **U** is $\mathbf{U}^T$, and the inverse of $\mathbf{U}^T$ is **U**.

Given **X**, when n and L are huge (e.g., n = thousands to millions and L = millions to billions), calculating **U** and **D** may seem impossible. However, this is rarely actually so. A number of clever methods exist which often make it possible to calculate the larger (i.e., upper left) diagonal entries in **D** and the corresponding left-most columns of **U** (which, as will be seen below, are usually all we need). There are highly parallelizable and accurate numerical linear algebra algorithms for calculating these elements (the principal eigenvalues and eigenvectors) of the SVD of matrices of the particular form $\mathbf{XX}^+$ (e.g., see [Diamantaras and Kung, 1996] for an introduction).

What the SVD of **C** shows is that when we multiply an **x** vector by **C** to form $\mathbf{Cx} = (\mathbf{UDU}^T)\mathbf{x}$, what we are first doing (i.e., in the product $\mathbf{U}^T\mathbf{x}$) is taking the dot product $\mathbf{u}_i \bullet \mathbf{x}$ of **x** with each of the vectors $\mathbf{u}_i$ and forming these scalar values into a column vector. This simply rotates (with possible reflections through the coordinate planes also) the vector **x** into a new vector $\mathbf{U}^T\mathbf{x}$ of the same length. The entries in this column vector $\mathbf{U}^T\mathbf{x}$ are the *coordinates* of **x** in the $\{\mathbf{u}_i\}_{i=1}^n$ basis for R^n. In the next product, $\mathbf{D}(\mathbf{U}^T\mathbf{x})$, each of these coordinates is multiplied by the corresponding d_i , i = 1, 2, …, n value. In the final product, $\mathbf{U}(\mathbf{DU}^T\mathbf{x})$, this *dilated coordinate vector* $(\mathbf{DU}^T\mathbf{x})$ is rotated back, using the reverse **U** of the original $\mathbf{U}^T$ rotation.

Given this geometrical analysis of **C**, the key observation is that if **C** is not going to be the identity matrix, the d_i dilation factors must not all be 1. In fact, what frequently happens is that the vast majority of the d_i factors turn out to be essentially zero (i.e., they are much smaller than the first few d_i factors, d_1, d_2 , …, d_p – where d_p is the last 'meaningfully large' value). What this tells us is that, essentially, the **x** vectors lie in the p-dimensional subspace of R^n spanned by the set of vectors $\{\mathbf{u}_i\}_{i=1}^p$. Note that the transformation of an **x** into this subspace can be carried out by multiplying it by $\mathbf{D\check{U}}^T$, where $\check{\mathbf{U}}$ is the matrix with columns 1 through p equal to $\mathbf{u}_1$, $\mathbf{u}_2$, …, $\mathbf{u}_p$, respectively; and the remaining columns are equal to the zero vector **0**. Scaling of the projected vector by **D** is used because this compensates for the weaker dependence of **x** on some of the subspace coordinates than others (note that this multiplicative scaling has no effect on components (p+1) through n, since these are all zero because of the form of $\check{\mathbf{U}}$).

The choice of the 'cutoff' dimension p is easier than it may seem. A typical process for choosing p is to calculate the MSE of data representation G(p) for the $\check{\mathbf{U}}^T$ built with the first p $\mathbf{u}_i$ vectors. This quantity is defined by

$$G(p) \equiv \frac{1}{L} \sum_{k=1}^{L} | \mathbf{x}_k - \mathbf{U}\mathbf{D}\breve{\mathbf{U}}^T \mathbf{x}_k |^2 \quad .$$

G(p) defines the mean squared error introduced over all of the data vector examples by using the projection of **x** into the p-dimensional subspace defined by the vectors $\mathbf{u}_1$, $\mathbf{u}_2$, …, $\mathbf{u}_p$, followed by scaling by **D**, and finally followed by the rotation of this projection back into the full n-dimensional space R^n. The way to analyze this quantity is to ask the question of how much effect this level of mean squared error (namely, G(p)) would have upon the application at hand. The value of p is typically chosen to be the smallest for which the effect of the MSE G(p) on the application would be negligible.

If the value of p derived by the above procedure is not much smaller than n, then it is probably best to forget the whole thing and proceed with the prescaled and z-scaled data vectors $\{\mathbf{x}_k\}_{k=1}^{L}$ as they are. However, if the value of p is significantly smaller than n; the data vectors $\{\mathbf{x}_k\}_{k=1}^{L}$ are transformed into the new set $\{\mathbf{D}\breve{\mathbf{U}}^T \mathbf{x}_k\}_{k=1}^{L}$ of data vectors, which, when the n-p zeros in the components p+1, p+2, …, n are truncated, lie in R^p. Whenever we need to transform one of these p-dimensional data vectors back to its original expression as an **x** vector we simply replace the previously truncated n-p zero components (placing the vector back in R^n) and then pre-multiply this n-dimensional vector by **U** (i.e., the n-dimensional data vector $\mathbf{D}\breve{\mathbf{U}}^T \mathbf{x}$ is restored to $\mathbf{U}\mathbf{D}\breve{\mathbf{U}}^T \mathbf{x}$; which will typically be very close to the original **x**). It is a subtle point worth noting that use of **D** in the transformation from n-dimensional space to the p-dimensional subspace yields a vector which is already approximately 'normalized' (since it need only be rotated by **U**, without length change, to bring it back to its original z-scaled form). Thus, these p-dimensional vectors are typically ready to be used in a perceptron without further prescaling or z-scaling.

When we can reduce the original n-dimensional data vectors to much-lower-dimensional p-dimensional data vectors, the perceptron model typically becomes much smaller, much easier to train, and more accurate.

The above process of finding the data subspace and then re-expressing the data vectors in terms of a basis within this subspace can sometimes provide another benefit: an insight into the structure of the data itself. This insight comes from examining the data subspace basis vectors $\mathbf{u}_1$, $\mathbf{u}_2$, …, $\mathbf{u}_p$. These are orthogonal vectors which form an approximate basis for the **x** data vectors. In effect, these vectors represent *principal linear orthogonal descriptive features* or *eigenfeatures* for the data vectors. $\mathbf{u}_1$ is the most important descriptive feature, $\mathbf{u}_2$ is the second most important descriptive feature, and so on. If there is some way to study these eigenfeatures they often reveal interesting insights into the problem being solved.

A typical example of the use of eigenfeatures would be where the $\{\mathbf{x}_k\}_{k=1}^{L}$ vectors are rasterized images of human faces (with each image the same size and each highpass-filtered face centered in the otherwise dark image and scaled as large as possible to just fit within a standard 'face-normalizing' oval). In this example, the images $\mathbf{u}_1$, $\mathbf{u}_2$, …, $\mathbf{u}_p$ are called *eigenfaces* [Kohonen, 1989]. These images can be viewed and, at least the first few, look like a face. The subsequent eigenfaces look like portions of the face that can either be emphasized (e.g., with a positive coordinate value) or de-emphasized (e.g., with a negative coordinate value). A few tens of such eigenfaces (i.e., for example, p might equal 30 in such a system) can be linearly combined (with the proper coefficients) to yield an image of almost anyone's face. In other words, the original **x** vectors (which might have been n = 1,000,000 -dimensional raw image brightness vectors which were then z-scaled) are reduced to vectors in a p = 30 -dimensional

subspace. This transforms an extremely difficult problem (directly inputting 1,000,000-dimensional image vectors of human faces into a perceptron) into a much simpler problem (inputting 30-dimensional eigenface coefficients into a perceptron).

Many kinds of real-world data vectors of large dimension n (face images, fixed-length time windows of sound, etc.) turn out to lend themselves to expression in terms of a relatively tiny number p of eigenfeature expansion coefficients. For some types of data (e.g., fingerprint images) this does not work as well as for others (i.e., a larger p value is required). However, it is a rare case when this methodology does not lead to a p which is substantially smaller than n.

For further information on this subject, refer to the **Perceptrons Aperitif** appended to the tail end of this document.

Appendix E: Pattern Classification Perceptrons

This Appendix presents the basics of pattern classification theory and then shows that perceptrons can be trained to approach a type of pattern classifier (a *Bayes* classifier) having optimum performance. The (often essential) pattern classification perceptron training techniques of *oversampling* and *normalization* are also described.

The starting point of pattern classification theory is to imagine a statistical source of *real-world objects* that are provided to us, one at a time, always randomly chosen in accordance with the fixed statistics of that source. Whenever we are given such an object, we place it into some kind of machine, which takes n *measurements* of the object; each assumed to be expressed in the form of a real-valued number. These measurement values, $x_1, x_2, \ldots, x_n$ (i.e., a point $\mathbf{x} \in R^n$) are then supplied to us by the machine as its output. The vector $\mathbf{x}$ is called a *measurement vector* or *pattern*. While the machine's operation may involve noise or error, it is assumed that its *statistical behavior* is *fixed* and does not vary with time (e.g., if we were to put each object of a large fixed collection of objects into the machine many times today we would get, for each such object in the collection, a collection of patterns x which would be statistically identical (i.e., converging to the same sample moment limits) to that which would have been obtained for that same object had we done this same experiment on some past day).

Each object that we are given is assumed to belong to one, and only one, of m *classes* of objects (which, therefore, are disjoint and exhaustive). However, note that there is no restriction that the patterns describing objects belonging to different classes are necessarily distinct. In other words, the measurement value outputs of our measurement machine may not always be sufficient to distinguish objects belonging to different classes (alternatively, the class inclusion decisions themselves may be "fuzzy" or noisy). It is assumed that underlying the pattern production process is a joint probability density $\rho(k, \mathbf{x})$ that gives the probability density of objects belonging to class k and having pattern $\mathbf{x}$. This joint density (which implicitly describes the characteristics of the object class statistics, of the object class inclusion decision process, and of the measurement machine) is assumed to exist and be well defined, but is almost always unknown and unknowable.

Assuming the existence of the joint density $\rho(k, \mathbf{x})$, the usual definitions of marginal and conditional probability can be applied to define the following mathematical quantities:

$p(k)$ = probability that a randomly chosen object belongs to class k $\equiv \int_{R^n} \rho(k, \mathbf{x})\, d\mathbf{x}$

$\rho(\mathbf{x})$ = probability density of the measurement vector $\mathbf{x}$ occurring $\equiv \sum_{\ell=1}^{m} \rho(\ell, \mathbf{x})$

$p(k \mid \mathbf{x})$ = probability that an object having measurement vector $\mathbf{x}$ belongs to class k $\equiv \dfrac{\rho(k, \mathbf{x})}{\rho(\mathbf{x})}$

$\rho(\mathbf{x} \mid k)$ = the probability density of the measurement vector $\mathbf{x}$ occurring, given that the object is known to belong to class k $\equiv \dfrac{\rho(k, \mathbf{x})}{p(k)}$.

Clearly, given these definitions, we have the following identity:

$$p(k \mid \mathbf{x})\, \rho(\mathbf{x}) = \rho(\mathbf{x} \mid k)\, p(k) \ , \qquad \text{(E1)}$$

since, by the above definitions, both sides are equal to $\rho(k, \mathbf{x})$. This probabilistic identity is known as *Bayes' law*.

The probability p(k) is termed the *a priori probability* of class k. This is the probability that a randomly chosen object will belong to class k. Obviously, $\sum_{k=1}^{m} p(k) = 1$, since each object is required to belong to one and only one class. The probability $\rho(\mathbf{x} \mid k)$ is termed the *class probability density function* of class k. $\rho(\mathbf{x} \mid k)$ is the probability density of finding a member of class k with measurement vector $\mathbf{x}$. $\rho(\mathbf{x})$ is the probability density of the measurement vectors $\mathbf{x}$ which occur across all randomly chosen objects without regard to class. Finally, $p(k \mid \mathbf{x})$ is termed the *a posteriori probability* of class k, given that the object in question has measurement vector $\mathbf{x}$.

Given these preliminaries regarding objects, patterns, classes, and related probability quantities, pattern classifiers are now defined and pattern classifier performance measurement is discussed.

A *pattern classifier* is a device which takes a pattern $\mathbf{x}$ (derived from an object of unknown class) as input and produces, as output, an estimate $C(\mathbf{x})$ of the class number to which the object belongs. Note that every classifier C partitions the pattern space R^n into m disjoint *decision regions* $R_1, R_2, \ldots, R_m$; where R_k is defined to be the set of points $\mathbf{x}$ for which $C(\mathbf{x})$ equals k.

The basic performance measure of a pattern classifier is its *average correctness* A. In other words, given a large number of objects chosen at random, how frequently does the classifier correctly classify them. To help develop a mathematical statement of this performance it is helpful to define the *performance function* $\lambda_C(\mathbf{x})$ of classifier C to be the probability that a vector $\mathbf{x}$ will be classified correctly by the classifier. [Note: This definition of performance assumes that all misclassifications are of equal importance to us. This definition can be altered to accommodate different levels of importance for misclassification of different classes of object. However, this is usually unnecessary. The discussion of *oversampling* below discusses some related issues.]

Using the function $\lambda_C(\mathbf{x})$, the *average correctness* A of classifier C is then defined to be

$$A = \int_{R^n} \lambda_C(x)\, \rho(x)\, dx \ .$$

What we would like to do is find a classifier $C^*(\mathbf{x})$ which provides the highest average correctness A. To do this, we use the a posteriori probability $p(k \mid \mathbf{x})$ defined above. Again, $p(k \mid \mathbf{x})$ is the probability that an object belongs to class k, given that the measurement vector (i.e., pattern) for that object is $\mathbf{x}$. A *Bayes' classifier* $C^*(\mathbf{x})$ produces, for each $\mathbf{x} \in R^n$, a class number output k such that

$$p(k \mid \mathbf{x}) \geq p(i \mid \mathbf{x})$$

for all i = 1, 2, …, m. It is now shown that the Bayes' classifier has the highest possible average correctness.

Start by noting that in the representation

$$A = \int_{R^n} \lambda_C(x)\, \rho(x)\, dx$$

what we want to do is have the classifier maximize the value of the integrand $\lambda_C(\mathbf{x})\,\rho(\mathbf{x})$ associated with each $\mathbf{x}$. This will maximize the integral. Of the two terms in this integrand, only the first depends on the classifier behavior. Thus, we need to maximize $\lambda_C(\mathbf{x})$ for each $\mathbf{x}$. However, the a posteriori probability $p(k\,|\,\mathbf{x})$ tells us, given $\mathbf{x}$, what the probability of the object being measured belonging to class k is. Thus, if our classifier output $C(\mathbf{x})$ is class i, this will be the correct answer with probability $p(i\,|\,\mathbf{x})$. In other words, the probability $\lambda_C(\mathbf{x})$ of the classifier giving the correct answer for a particular $\mathbf{x}$ is equal to $p(C(\mathbf{x})\,|\,\mathbf{x})$. So, we can maximize $\lambda_C(\mathbf{x})$ for each $\mathbf{x}$ by setting $C(\mathbf{x})$ equal to the class having the *maximum a posteriori probability* $p(k\,|\,\mathbf{x})$. This is what the Bayes' classifier does, thus it is optimal.

The implementation of a Bayes' classifier by a pattern classification perceptron is now discussed.

Until recently, it was not usually possible to build a Bayes classifier for solving a practical problem. However, this changed in 1990 with the announcement of a neural network method developed independently by Wan and by Ruck and his colleagues (their papers appeared in the same issue of the same journal!) [Wan 1990, Ruck et al 1990]. These papers proved the following theorem:

Theorem [Pattern Classification Perceptron Theorem (Wan 1990, Ruck et al 1990)]:

Given randomly chosen examples $\{(\mathbf{x}_1,\mathbf{y}_1), (\mathbf{x}_2,\mathbf{y}_2), \ldots, (\mathbf{x}_k,\mathbf{y}_k), \ldots\}$ of correctly classified patterns (where the $\mathbf{y}$ vectors give the class of pattern $\mathbf{x}$ in the form of a 1-out-of-m code, as in Section 1) from a real-world source, construct a perceptron (via the training and architecture adjustment process described in Section 7) to approximate the regression function of the joint probability density $\rho(k,\mathbf{x})$ from which the above vector pairs were obtained as randomly chosen examples. In the limit as the mean squared error of this approximation approaches its minimum value A [**NOTE**: this minimum will not necessarily be a small value – see Section 4], the k^{th} output of the multilayer perceptron will approach the class a posteriori probability $p(k\,|\,\mathbf{x})$ of class k.

Proof: A perceptron minimizes the mean squared error $E[|\mathbf{y}\text{-}\mathbf{y}'|^2]$ and thus, by Theorem 2, it converges to the regression function $\mathbf{r}(\mathbf{x}) = E[\mathbf{y} \mid \mathbf{x}]$, the average of the $\mathbf{y}$ vectors which occur for a particular $\mathbf{x}$. For the 1-out-of-m code $\mathbf{y}$ vectors used for coding the pattern class of the $\mathbf{x}$ vectors, the i^{th} component of the regression function, $E[y_i \mid \mathbf{x}]$, is equal to the average fraction of the time an object with measurement vector $\mathbf{x}$ belongs to class i. This is exactly the a posteriori probability $p(i \mid \mathbf{x})$ of that class. ■

Thus, to implement a Bayes' classifier, all we need to do is construct a perceptron with near-minimum mean squared error (by training on randomly chosen examples $(\mathbf{x},\mathbf{y})$ – see below for details) and then, for each new pattern to be classified, simply choose that class k whose associated output (which, by the above theorem, will approximate $p(i \mid \mathbf{x})$, the a posteriori probability of class k) is largest. This will be the class that we assign this unknown pattern to (i.e., this will be the class number output of our perceptron pattern classifier). Notice that for points $\mathbf{x}$ where multiple classes have the same maximum a posteriori probability it doesn't matter which of these classes the classifier chooses: it will still have the highest possible average correctness of classification. Except for the choices made at such unusual ambiguous points, the Bayes' classifier decision output is unique.

Although minimizing the mean squared error of a classification perceptron does indeed lead to each of the perceptron's outputs being equal to the a posteriori class probability of the corresponding class, the convergence rate of a real perceptron training process will, in general, not be equal for all outputs. In fact, the convergence rate of each output to its class a posteriori probability is roughly a direct function of the percentage of the number of examples of its class that are presented during training. However, since the presentation of examples during training of a 'Bayes' classifier perceptron' must be random, these percentages (which correspond to the class a priori probabilities p(k)) will not, in general, be equal. Thus, there is almost always a conflict between unequal class a priori probabilities and equally fast convergence of each class output to it's a posteriori probability function. An example of one way to reconcile this conflict is now presented.

A very useful observation about the Bayes classifier perceptron is to note that Equation E1 above can be re-written as

$$p(k \mid \mathbf{x}) = \frac{\rho(\mathbf{x} \mid k)\, p(k)}{\rho(\mathbf{x})} \quad . \qquad \text{(E2)}$$

This shows us that we can alternatively interpret the outputs of our perceptron as the quantities on the right hand side of this equation. Note that, in this right-side expression, the a priori probability p(k) appears explicitly and that the other portions of this expression do not in any way depend upon p(k). This allows a very important technique called *oversampling* to be used. The basic idea of oversampling is to build a phony data source for training the classification perceptron and then make compensating adjustments to the classifier perceptron after training so that it will function well for the real-world pattern classification problem.

The first step in oversampling is to build m training sets, with training set i (i = 1, 2, … ,m) consisting of randomly chosen (**x**,**y**) examples from class i. Often, these individual class training sets will not be the same size, since we often possess fewer examples of some classes than others. These size differences do no harm, as long as the total size of the smallest class training set is sufficiently large for classifier perceptron training with adequate performance on that class. Of course, random and sufficiently large training test, validation, and verification sets must also be created for each class and all of the **x** pattern data must be pre-scaled and z-scaled (of course, the 1-out-of-m code y vectors must be used unaltered).

Once the m class training sets have been created, we then proceed with training and architecture adjustment of the classification perceptron. At each training step we first chose one of the m classes uniformly at random and then choose the next training example in sequence from the training set of that class (the same example selection procedure is used during training test, validation, and verification). When the last example in an example set is used, the next sample used from that set is the one at its beginning. Note that this process will cause the examples of the smaller class training sets (these are often the classes we are most interested in spotting correctly in the application) to be used more times during training than those of the larger training sets. This is the origin of the term *oversampling* for this technique. Other than this method of choosing the examples to be used, training and architecture adjustment proceed just as described in Section 7. Oversampling supports a more or less equal rate of convergence in each class output component to its oversampled a posteriori probability function.

Once the classification perceptron has converged and its performance meets our classification accuracy goals (note that testing is on the basis of classification accuracy with all class a priori probabilities equal to 1/m), this trained classification perceptron is then *normalized* by multiplying all of the weights in output unit k by the factor

$$m \; p(k)$$

for k = 1, 2, … , m. Normalization causes the right side of equation E2 – which had been equal to

$$\frac{\rho(\mathbf{x}\,|\,k)}{m\;\rho(\mathbf{x})}$$

during training – to be restored to its proper value

$$\frac{\rho(\mathbf{x}\,|\,k)\,p(k)}{\rho(\mathbf{x})} \quad . \qquad \text{(E2)}$$

Once normalized, the trained perceptron is then ready for use in the real-world classification problem.

It is common misconception that the as-trained, un-normalized classification perceptron will work well in the practical application. However, this is false, as the a priori probabilities have a significant effect on the classification accuracy of the system. Normalization is essential in order to achieve the maximum classification accuracy A.

Finally, it is often the case that classification errors have different levels of *relative importance*, depending on the class on which the error is made. If these performance concerns can be expressed as specific multiplicative positive *relative importance* factors e_1, e_2, … , e_m for the m classes (where $\sum_{j=1}^{m} e_j = 1$ and where the larger the value of e_k the more important it is not to make an error on that class) then all we need to do is multiply each output of the classifier by the corresponding e_k value and then take the larger of these products. Of course, the price of doing better on more important classes is making more classification mistakes on less important classes.

Appendix F: Perceptron Input and Output Coding Techniques

In Appendix E it was shown that pattern classification perceptrons need to use 1-out-of-m codes for their output; in order to yield the desired a posteriori probability estimates. This Appendix discusses other techniques for the representation of specific kinds of data in perceptron inputs and outputs.

As in pattern classification, the rate of convergence of a perceptron to a desired regression function can be dramatically increased (or even brought from the realm of the unfeasible to the realm of the feasible) if the proper coding of data in the (**x**,**y**) vectors is used. The two cases which will be considered here are: inherently real-valued data and inherently categorical data.

An *inherently real-valued variable* (e.g., a person's age or the brightness of a pixel) is data which can vary continuously or discretely over a real number interval and where the input-output function of the perceptron can be expected to, in general, be insensitive to small perturbations of this variable. Inherently real-valued variables should be pre-scaled and z-scaled and then used directly for input to and/or output from the perceptron.

An inherently categorical variable (e.g., female / male, Lexus / Toyota / Nissan / Chevrolet / … / Rolls Royce) typically has the property that its different values do not have any order relationship among them that is meaningful for the perceptron being built. Since the outputs of the hidden neurons of perceptrons respond continuously, it is important that each categorical variable be coded in a way that fits this type of response. The dominant technique for doing this (which almost always works very well) is to express categorical variables with K categories or classes using the same 1-out-of-K code used in pattern classification class coding. Thus, for example, a single categorical variable having K categories would be represented by creating K new components (for the **x** or **y** vector) and coding the variable's value by setting the appropriate one of these to 1.0 and the others to 0.0. Categorical variable inputs are not pre-scaled, nor z-scaled.

If K is large, coding a categorical variable using a 1-out-of-K code can make the effected **x** or **y** vector's dimension much larger. This is usually not a problem, even though the number of hidden units often must significantly exceed the dimensionalities of the input and output vectors combined. Training time typically does not go up dramatically with the number of categories. A classic example of the successful use of this technique is the *Net Talk* speech synthesis system [Sejnowski and Rosenberg, 1987], which had seven categorical inputs (each a text symbol consisting of 29 possible alphabetic characters and punctuation marks) and one categorical output (with each category representing one of about 50 distinct phonetic sounds).

Recommended for Further Reading

Arbib MA (ed) (1995) The Handbook of Brain Theory and Neural Networks. Cambridge, Massachusetts: MIT Press.

Fine TL (1999) Feedforward Neural Network Methodology. New York: Springer-Verlag.

Haykin S (1999) Neural Networks: A Comprehensive Foundation, Second Edition. Upper Saddle River, New Jersey: Prentice-Hall.

References

Abu-Mostafa YS (1995a) Hints. *Neural Computation* **7**: 639-671.

Abu-Mostafa YS (1995b) Machines that learn from hints. *Scientific American* **272**: 64-69.

Amari S (1967) A theory of adaptive pattern classifiers. *IEEE Transactions on Electronic Computers* **EC-16** (3): 299-307.

Anderson JA and Rosenfeld E (1988) Neurocomputing: Foundations of Research. Cambridge, Massachusetts: MIT Press.

Anderson JA and Rosenfeld E (1998) Talking Nets. Cambridge, Massachusetts: MIT Press.

Arbib MA (ed) (1995) The Handbook of Brain Theory and Neural Networks. Cambridge, Massachusetts: MIT Press.

Bishop CM (1995) Neural Networks for Pattern Recognition. New York: Oxford University Press.

Browder W (1996) Mathematical Analysis: An Introduction. New York: Springer-Verlag.

Casasent D and Barnard E (1990) Adaptive clustering optical neural net. *Applied Optics* **29**: 2603-2615.

Chen AM, Lu H-m and Hecht-Nielsen R (1993) On the geometry of feedforward neural network error surfaces. *Neural Computation* **5**: 910-927.

Cristianini N and Shawe-Taylor J (2000) An Introduction to Support Vector Machines. Cambridge, UK: Cambridge University Press.

Davidson J (1994) Stochastic Limit Theory. Oxford, UK: Oxford University Press.

Denoeux T and Lengelle R (1993) Initializing back propagation networks with prototypes. *Neural Networks* **6**: 351-363.

Devroye L, Györfi L and Lugosi G (1996) A Probabilistic Theory of Pattern Recognition. New York: Springer-Verlag.

Diamantaras KI and Kung SY (1996) Principal Component Neural Networks. New York: Wiley.

Duda RO and Hart PE (1973) Pattern Classification and Scene Analysis. New York: Wiley.

Dudley RM (1999) Uniform Central Limit Theorems. Cambridge, UK: Cambridge University Press.

Fine TL (1999) Feedforward Neural Network Methodology. New York: Springer-Verlag.

Fishman GS (1996) Monte Carlo: Concepts, Algorithms, and Applications. New York: Springer-Verlag.

Fukumizu K and Amari S (2000) Local minima and plateaus in hierarchical structures of multilayer perceptrons. *Neural Networks* **13**: 317-327.

Haykin S (1999) Neural Networks: A Comprehensive Foundation, Second Edition. Upper Saddle River, New Jersey: Prentice-Hall.

Hebb DO (1949) The Organization of Behavior: A Neuropsychological Theory. New York: Wiley.

Hecht-Nielsen R (1990) Neurocomputing. Reading, Massachusetts: Addison-Wesley.

Huber PJ (1996) Robust Statistical Procedures. Philadelphia, Pennsylvania: SIAM.

Kohonen T (1989) Self-Organization and Associative Memory, Third Edition. Berlin: Springer.

Kůrkova V and Kainen PC (1994) Functionally equivalent feedforward neural networks. *Neural Computation* **6**: 543-558.

Kushner HJ and Yin GG (1997) Stochastic Approximation Algorithms and Applications. New York: Springer.

McCulloch WS and Pitts W (1943) A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics* **5**: 115-133.

Minsky M and Papert S (1969) Perceptrons. Cambridge, Massachusetts: MIT Press.

Niederreiter H (1992) Random Number Generation and Quasi-Monte Carlo Methods. Philadelphia, Pennsylvania: SIAM.

Nilsson NJ (1965) Learning Machines. New York: McGraw-Hill.

Poggio T and Girosi F (1993) Learning algorithms and network architectures. In: Aertsen A (ed) *Brain Theory,* pp. 29-46. Amsterdam: Elsevier.

Poggio T and Girosi F (1990) Regularization algorithms for learning that are equivalent to multilayer networks. *Science* **247**: 978-982.

Rosenblatt F (1962) Principles of Neurodynamics. Washington DC: Spartan Books.

Ruck D, Rogers SK, Kabrisky M, and Oxley ME (1990) The multilayer Perceptron as an approximation to a Bayes optimal discriminant function. *IEEE Transactions on Neural Networks* **1**: 296-298.

Rumelhart DE, Hinton GE and Williams RJ (1986) Learning representations by back-propagating errors. *Nature* **323**: 533-536.

Rumelhart DE and McClelland JL (eds) (1986) Parallel Distributed Processing: Explorations in the Microstructure of Cognition, I & II. Cambridge, Massachusetts: MIT Press.

Schmeisser H-J and Triebel H (1987) Topics in Fourier Analysis and Function Spaces. Chichester, UK: Wiley.

Sejnowski T and Rosenberg CR (1987) Parallel networks that learn to pronounce English text. *Complex Systems* **1**: 145-168.

Shepanski JF and Macy SA (1988) Teaching artificial neural systems to drive: Manual training techniques for autonomous systems. In: Anderson DZ (ed) *Proceedings of the 1987 Neural Information Processing Systems Conference,* pp. 693-700. New York: American Institute of Physics.

Steinbuch K (1961) Automat und Mensch. Berlin: Springer-Verlag.

Steinbuch K and Piske UAW (1963) Learning matrices and their applications. *IEEE Transactions on Electronic Computers* **12**: 846-862.

Strang G (1988) Linear Algebra and its Applications, Third Edition. Fort Worth, Texas: Harcourt.

Sussmann HJ (1992) Uniqueness of the weights for minimal feedforward nets with a given input-output map. *Neural Networks* **5**: 589-593.

Telfer B and Casasent D (1991) Minimum-cost Ho-Kashyap associative processor for piecewise-hyperspherical classification. In: *Proceedings of the 1991 International Joint Conference on Neural Networks* **2**: 89-94. Piscataway, New Jersey: IEEE Press.

Tsypkin YZ (1973) Foundations of the Theory of Learning Systems. New York: Academic Press.

Vapnik VN (2000) The Nature of Statistical Learning Theory, Second Edition. New York: Springer-Verlag.

Wan E (1990) Neural network classification: A Bayesian interpretation. *IEEE Transactions on Neural Networks* **1**: 303-305.

Widrow B and Hoff ME (1960) Adaptive switching circuits. In: *1960 IRE WESCON Convention Record,* pp. 96-104. New York: IRE.

Zhang F (1999) Matrix Theory. New York: Springer.

Perceptrons Aperitif

The Geometrical Structure of Real-World Data

Notes for
ECE-173 / ECE-270

UCSD

25 February 2002

Robert Hecht-Nielsen

1. Data Sources

A useful model is to view each *source* of real-world data (imagery, text, sound, database records, etc.) as a collection of *data items* (each expressed as a vector **x** in R^m , m-dimensional Euclidean space) belonging to a delimited subset $A \subset R^m$ (A is assumed to be a diffeomorphism of the unit sphere). A probability density function $\rho : A \subset R^m \to R^+$ (assumed to be smooth), called the *a priori density* of the source, defines how frequently we would expect a particular data item **x** to appear in large volumes of natural data from the source.

2. Outline

The document that follows, **Perceptrons**, looks at mappings $f : A \subset R^m \to B \subset R^n$ between data sources A and B. In fact, we will show that all such mappings can be expressed as

$$y = f(x) = V[\tanh(Ux)] \ ,$$

where U and V are matrices. This structure is called a *perceptron neural network*. Such mappings can implement a vast variety of information processing operations of great practical value. However, in this document, we will only consider a single data source and look at ways of understanding its geometrical and probabilistic structure. In order to proceed, we will need some results from elementary linear algebra. These are sketched next (for further details see a linear algebra text or one of the many 'matrix handbooks').

3. Diagonal and Orthogonal Matrices and the SVD

An m x n matrix $D = [d_{ij}]$ is called *diagonal* if $d_{ij} = 0$ whenever $i \neq j$. A p x p matrix P is called *orthogonal* if $P^{-1} = P^T$ (= $[p_{ji}]$, the *transpose* of t). Note that this implies that $PP^T = P^TP = I_p$ (where I_p is the p x p identity matrix – note: we will suppress the dimension subscript, since it is almost always obvious). An example of an orthogonal matrix is the identity matrix I (which is also diagonal). Orthogonal matrices all have a determinant of +1 or –1. Also, all of the rows and columns of an orthogonal matrix P are unit-length vectors. The *row vectors* of P are all pairwise mutually perpendicular (p-dimensional) vectors; as are all of the *column vectors*. These are valuable, but non-obvious, facts that took a long time to discover.

An amazing mathematical result (which was not discovered until about 30 years ago) is that *every* m x n matrix U can be expressed as the matrix product of an m x m orthogonal matrix P, an m x n diagonal matrix D, and an n x n orthogonal matrix Q. This can be expressed as:

$$U = PDQ^T \quad .$$

The matrices P, D, and Q are not unique. We call such a re-expression of a matrix U a *singular valued decomposition* (SVD). Note that if Q is orthogonal then so is Q^T, because $[S^T]^T = S$ and so $[Q^T]^T\ Q^T = Q\ Q^T = Q^T\ Q = Q^T\ [Q^T]^T = I$.

There are many additional facts known about the P, D, and Q matrices that arise in an SVD of the matrix U. We will need only one of these; which is that the diagonal entries of the m x n matrix $D = [d_{ij}]$ satisfy the following inequalities (remember that, since D is diagonal, $d_{ij} = 0$ whenever $i \neq j$)

$$d_{11} \geq d_{22} \geq d_{33} \geq \ \ldots\ \geq d_{rr} \geq\ \ldots\ \geq d_{\min(m,n)\ \min(m,n)} \geq 0 \quad ,$$

where d_r (we commonly suppress repeating the index) is the last positive entry along the diagonal of D (all the subsequent values, if any there are, are zero). The number r is called the *rank* of the matrix U. These diagonal entries in D are called the *singular values* of the matrix U.

4. The Geometry of Orthogonal and Diagonal Matrices

If we are considering a linear transformation y = Ux, we can use an SVD to re-express this as $y = PDQ^T\ x = P(D(Q^T\ x))$ (because matrix multiplication obeys the *associative law*). Thus, we see that the linear transformation can be decomposed as three separate linear transformations in a row: multiplication by an orthogonal matrix, multiplication by a diagonal matrix of rank r, and multiplication by another orthogonal matrix. Now, we consider the geometrical effect of each of these transformations individually.

First, it is easy to show that premultiplication of a p-dimensional vector z by an orthogonal p x p matrix P does not change the length of the vector. To show this we need the facts that if a and b are p-dimensional column vector matrices, then

$$a \cdot b = a^Tb$$

and that

$$[AB]^T = B^T\ A^T \quad .$$

Then, recalling that the square of the *length*, or *magnitude*, of a vector a is given by

$$| a |^2 = a \cdot a$$

we get

$$| Pz |^2 = (Pz)^T (Pz) = z^T P^T Pz = z^T I z = z^T z = | z |^2 \quad .$$

So, premultiplying any p-dimensional vector z by an orthogonal p x p matrix P yields another p-dimensional vector Pz which has the same length as z. A geometrical transformation from R^p to R^p which does not change the lengths of any vectors is called an *isometry* or generalized *rotation*. Thus, an orthogonal matrix transformation carries out a geometrical rotation. In fact, it can be shown that *any* possible rotation can be carried out by an orthogonal matrix.

Premultiplying an n-dimensional vector z by an m x n diagonal matrix D of rank r has a very simple effect. The vector Dz has its first r components equal to, respectively,

$$z_1 d_{11}, z_2 d_{22}, \ldots, z_r d_{rr} \quad ,$$

and all of its later components (i.e., component numbers (r+1), (r+2), … , (n-1), n , if any there be) are zero. We call such a geometrical transformation a *dilatation*. In effect, what happens is that the first r components of z are multiplied by fixed positive constants and the remaining components (if r < n) are all set equal to zero.

Thus, any linear mapping y = Ux = PDQ^T x can be viewed as the composition of three successive geometrical mappings: a rotation, a dilatation, and another rotation.

A useful fact to note is that if the matrix U is square its SVD is U = PDP^T ; i.e., for a square matrix Q = P. In this case, the geometrical rotation P is the exact reverse of the rotation P^T . In this case, if we let the n columns of the matrix P be denoted by $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \ldots, \mathbf{p}_n$ (remember, they are mutually orthogonal unit vectors) then note that

$$U \mathbf{p}_i = PDP^T \mathbf{p}_i = PD \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} = P \begin{bmatrix} 0 \\ \vdots \\ d_i \\ \vdots \\ 0 \end{bmatrix} = d_i \mathbf{p}_i \quad .$$

This shows that the columns $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3, \ldots, \mathbf{p}_n$ of P are the *eigenvectors* of U and that the diagonal entries $d_1, d_2, \ldots, d_n$ of D are the *eigenvalues* of U (note: *eigen* in German means *characteristic* and implies *unchanged* or *invariant*). The value of this will become clear below.

5. The Pseudoinverse and Some of Its Properties

Given any SVD $U = PDQ^T$ of *any* m x n matrix U, we define the *pseudoinverse* U^+ of U to be the n x m matrix $U^+ = QD^+P^T$, where the matrix D^+ is an n x m diagonal matrix having its first r diagonal entries equal to

$$(d_{11})^{-1},\ (d_{22})^{-1},\ (d_{33})^{-1},\ \ldots\ ,(d_{rr})^{-1}\ ,$$

and all other entries equal to zero (note that since all of the first r diagonal entries of D are positive, so will the first r entries of D^+ , namely, their inverses).

Although it is not obvious (since the SVD is not unique), the pseudoinverse (which was not discovered until about 1955) *is* unique. Each matrix U has one and only one pseudoinverse U^+ . Further, if U has an inverse U^{-1} (i.e., it is a square matrix and has a non-zero determinant; i.e., is *non-singular*), then its pseudoinverse is its inverse (i.e., in this case, $U^+ = U^{-1}$).

Although, in general, UU^+ does not equal the identity matrix (nor does U^+U), the pseudoinverse of a matrix does have several very useful properties. Some valuable facts are listed below:

1. rank U = rank U^+
2. $(U^+)^+ = U$
3. $UU^+U = U$
4. $U^+UU^+ = U^+$
5. $(U^+U)^T = U^+U$ and $(UU^+)^T = UU^+$ (i.e., U^+U and UU^+ are *symmetric* matrices)

6. The Pseudoinverse and Minimum-Error Linear Equation Solutions

Given any general matrix equation of the form AX = B (called a *linear equation*), where A is a known m x n matrix, X is an unknown n x p matrix, and B is a known m x p matrix, the problem is to find the matrix X of minimum *norm* which minimizes the norm of the *error matrix* E

$$E = AX - B\ ,$$

(where the *norm* of a matrix E is the value of d_{11} in any of its singular valued decompositions $E = PDQ^T$ (d_{11} turns out to be the same in any SVD of E)). This norm-minimizing X is called the *minimum squared error* (MSE) "solution" to the problem AX = B. No matter what the matrices A and B are, the MSE solution X always exists and is unique. The MSE solution X turns out to be an extremely valuable quantity. The amazing fact is that this MSE solution is, in all cases, given by:

$$X = A^+B\ .$$

It is instructive to look at the simple case Ax = b, where X and B are, respectively, simply n-dimensional and m-dimensional vectors. It is easy to see that a matrix equation of this form Ax = b represents a set of m equations in n unknowns (the components of x). If m < n then there are too few equations to fully

define the components of x and Ax = b typically has an infinite number of solutions. In this case, the MSE solution $x = A^{+}b$ is the solution vector with the shortest length.

If m = n and Ax = b has a unique solution, then A turns out to be non-singular and $x = A^{+}b = A^{-1}b$.

If m > n (i.e., more equations than unknowns) there is usually not any vector x that satisfies all m equations simultaneously. In this case, the MSE solution $x = A^{+}b$ is the shortest vector x that minimizes the length of the *error vector* e = Ax – b.

An interesting fact is that Karl Friedrich Gauss invented the MSE method in the late 1790s as the solution to an astronomy problem. The newly discovered asteroid *Ceres* had been observed by astronomers only a few times (insufficient to calculate its orbit) when it became invisible for six months in the sunlight. Gauss (who was in his early 20s at the time) realized that he could greatly aid the launching of his career if he could somehow predict the orbit of Ceres based upon these incomplete observations and announce where the asteroid would reappear in the sky. He invented the MSE method, applied it to find the orbit with the least eccentricity that fit the observations, publicly announced his prediction (with maneuvers to ensure widespread dissemination of the story), his prediction turned out to be correct, and this instant fame and acclaim led to inauguration of generous support of his work. Even in the 18th century, well-managed PR was a valuable component of scientific career management.

7. The Nature of Real-World Data, Principal Components, and Trimming of Singular Values

Let us consider a set of L real-world m-dimensional data vectors $\{\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_L\}$ drawn at random from some data source ρ (note that we typically use the a priori probability density of a source as the name for that source). We begin by calculating the mean of this sample $\bar{x} \equiv \frac{1}{L}\sum_{i=1}^{L}\hat{x}_i$ (which we assume is so large and statistically representative of the source that we can use $\bar{x}$ as the actual source mean). Because matrix analysis of the geometry of sources is always centered on the origin of the coordinate system, we must place the center of our data source there. This is easily done by subtracting the source mean $\bar{x}$ from each data item; yielding a new set of data vectors $\{x_1, x_2, \ldots, x_L\}$; where $x_i = \hat{x}_i - \bar{x}$. [**CRITICAL NOTE**: It is ESSENTIAL that we remember to add this mean $\bar{x}$ back in before ever trying to use a data item and to subtract it before using the matrix analysis results presented below. No further warning of this will be given!]

These data items could be almost anything: images, instrumentation readings, sounds, etc. For fun, let us imagine that these are L = 100,000,000 pictures of different, uniformly randomly chosen, human faces with their nose tips centered in 128 x 128 pixel, 8-bit grayscale, digital panchromatic images with each face occupying the whole image width. Thus, each data vector $\hat{x}$ consists of m = 128 x 128 = 16,384 components, each an integer grayscale value between 0 (completely dark pixel) and 255 (completely light pixel). Let L X be the m x L matrix having the L x_i vectors (in order, from i = 1 to i = L) as its columns. Thus, the matrix X has each x_i vector component divided by L.

From the above theory of the pseudoinverse we know that the (16,364 x 100,000,000) matrix X has a pseudoinverse X^{+} and that it is unique. We also know (from property 3 of the pseudoinverse) that

$$(XX^{+})X = X \quad .$$

However, if we take an SVD of X we can write $X = PDQ^T$ and $X^+ = QD^+P^T$ and this relationship becomes

$$(PDQ^T\ QD^+P^T)X = X \ .$$

Recalling that Q is orthogonal, $Q^T Q = I_L$ and so this relationship becomes

$$(PDD^+P^T)X = X \ ,$$

where the m x m matrix DD^+ has r 1's down its diagonal and all other entries zero and where r is the rank of the matrix X (the non-zero entries of DD^+ are all 1 because each diagonal entry in D^+ , $(d_{ii})^{-1}$, is the inverse of the corresponding entry, d_{ii} , of D).

One of the most amazing, mysterious, and valuable facts about real-world data is that the rank of X, r, very very often (but not always) turns out to be much smaller than m. For example, in the case of faces, r turns out to be around 30 [2]. Thus, if we consider the i^{th} "face image" vector x_i to be an m x 1 column vector, the m x 1 column vector $DD^+P^T x_i$ has only about 30 non-zero numbers in it (the first thirty). Thus, even though a face image contains 16,384 numbers, the actual information in the image can be accurately expressed with only 30 numbers. Clearly, if we premultiply the very sparse vector $DD^+P^T x_i$ by P we recover x_i exactly, without error. Thus, somehow, the SVD of X contains profoundly valuable information about the deep structure of these face images.

To get at this profound information, notice that the transformation $P^T x_i$ is, by the results of the last section, nothing but a rotation of x_i . Premultiplication of this rotated vector by DD^+ then has the effect of "zeroing out" all but the first thirty of its rows (which are left intact). If we carefully examine the product $P^T x_i$ we see that these first thirty components of this product vector (the only ones that matter) are obtained by taking the inner products of the first thirty rows of P^T with the vector x_i . However, the first thirty rows of P^T are the first thirty columns of P. Of course, in the general (rank r) case, these critical vectors would be in the first r columns of P (eigenvectors of X). These r column vectors (of unit length and all mutually orthogonal, as discussed above) are called the *principal components* of the matrix X. Notice that we could replace all of the other columns of P with arbitrary vectors and it would make no difference at all (since all remaining components of the product $P^T x_i$ are zeroed out by DD^+ anyway).

As an aside, note that if some of the higher singular values are small, but not zero, it is often OK to simply force them to zero (let's call this revised matrix $\check{D}$). The error that this causes in the replication of vector x_i can be assessed by calculating $| x_i - P\check{D}\check{D}^+P^T x_i |^2$ for each of the x_i vectors. If this squared error is suitably small (e.g., the face image $P\check{D}\check{D}^+P^T x_i$ is very little different from x_i), then this even smaller *reduced rank SVD* (which will have the new, smaller, number r of singular values and principal components) can be used instead of the original SVD. In effect, we have two linear mappings: one ($\check{D}\check{D}^+P^T x_i$) from R^m to R^r and another (essentially, the m x r matrix consisting of the first r columns of the matrix P; the principal components of X) back from R^r to R^m.

One of the key insights that can be gained from this kind of analysis is to carefully examine the principal components themselves (e.g., since they are m-dimensional unit vectors they can be treated as images – although their values are not guaranteed to be on the right scale; for example, some might be negative). These vectors define the *features* of the data vectors that are the richest in descriptive information. Time

spent analyzing principal components often yields valuable insights into the structure of the associated real-world data. This is often referred to as *principal components analysis* (PCA).

Some PCA examples: Eigenfaces (30 components), ocean thermoclines (three components), local foveal visual data (a few hundred components), local sound data (a few hundred components), luggage scanner data, SEXNET.

8. Calculating Principal Components for Real Data Sources

Obviously, if we were to form the (16,364 x 100,000,000) matrix X we could, in principle, submit this matrix to a mathematical software package, say **Mathematica™**, and simply ask it to carry out SVD and give us the singular values and the columns of P. The problem is, even if our computer were large enough to handle this terabyte matrix, the calculations might require many years to carry out. Thus, we must find a more practical method for discovering these quantities (the non-small singular values and the corresponding columns of P).

Consider the following. The matrix XX^T is an (m x m) matrix that has the form $PDQ^T\ QD^TP^T = PDD^TP^T$. The matrix DD^T is an (m x m) square diagonal matrix with non-zero diagonal entries $d_1^2, d_2^2, \ldots, d_r^2$. All other entries in this matrix are zero.

Since the columns of P form a basis for R^m (the space in which the images live), let us consider what happens when we multiply an arbitrary unit-length image vector y (containing at least some component in the direction of the first column vector of P) by the matrix XX^T. To see what happens, let us express y in terms of the basis formed by the columns of P

$$y = t_1\, p_1 + t_2\, p_2 + \cdots + t_m\, p_m \quad .$$

Then we get

$$\begin{aligned} XX^T\, y &= t_1\, XX^T\, p_1 + t_2\, XX^T\, p_2 + \cdots + t_m\, XX^T\, p_m \\ &= t_1\, PDD^TP^T\, p_1 + t_2\, PDD^TP^T\, p_2 + \cdots + t_m\, PDD^TP^T\, p_m \quad . \\ &= t_1\, d_1^2\, p_1 + t_2\, d_2^2\, p_2 + \cdots + t_m\, d_m^2\, p_m \end{aligned}$$

What this shows is that when we multiply y by the matrix XX^T the component of y that points in the direction of the first principal component has its magnitude increased relative to that of the other components (since the singular values are arranged in decreasing size). If we were to take the vector that we get as the result of this operation and again normalize it to unit length (without changing its direction) by simply dividing it by its magnitude, this new unit vector would be rotated closer towards p_1, relative to the original vector y. Notice that the components of the output vector beyond r are all zero, since the singular values are zero. It is easy to see that if we continued doing this recursively the eventual result would be convergence of the unit vector to p_1. Note that once we have discovered p_1 by this means, we can simply multiply it by XX^T one more time and the output will be $d_1^2\, p_1$. Thus, we will also know the value of the first singular value of X.

This "adaptive" way of finding the first principal component and singular value is better than trying to do an SVD on X, but it still requires us to deal with the terabyte matrix XX^T. So, another idea is used instead. This is based upon the fact that we can write

$$XX^T = \frac{1}{L}\left[x_1 x_1^T + x_2 x_2^T + \cdots + x_L x_L^T\right] \quad .$$

[**NOTE**: From this formula it is easy to see that the matrix XX^T will converge to some average value as L increases (again assuming that the x data vectors are chosen from the source at random in accordance with ρ). This limiting matrix is called the *covariance matrix* of the source. It is actually the singular values of the covariance matrix which matter most.]

If we assume that these sample data vectors are statistically representative of the whole source, then we could carry out a simplified version of the above process using a mathematical technique called *stochastic approximation*. In this process we choose an initial unit-length vector y_0 uniformly at random in R^m. We then choose an image vector x from the source at random and calculate the vector y_1 as follows

$$y_1 = \frac{y_0 + \beta (x\, x^T)\, y_0}{\left| y_0 + \beta (x\, x^T)\, y_0 \right|} = \frac{y_0 + \beta\, x\, (x^T\, y_0)}{\left| y_0 + \beta\, x\, (x^T\, y_0) \right|} \quad ,$$

where β is a small number (say, 0.10). This is *Oja's principal component learning law* [1]. We then continue iteratively with this process to derive y_2, and then y_3, and so on, until the vector y stops changing (in which case it is approximately equal to p_1), via the formula (where, on each iteration, a new image x is chosen randomly in accordance with ρ)

$$y_k = \frac{y_{k-1} + \beta_k\, x\, (x^T\, y_{k-1})}{\left| y_{k-1} + \beta_k\, x\, (x^T\, y_{k-1}) \right|} \quad ,$$

where the *learning rate* β_k is recalculated at the beginning of each iteration (after the first, where the initial $\beta_1 = \beta$) in accordance with the formula

$$\beta_k = \frac{\beta_{k-1}}{\gamma + (x^T\, y_{k-1})^2\, \beta_{k-1}} \quad ,$$

where γ is a constant chosen by the user to ensure that the value of β_k slowly decreases.

Note that the matrix product $x^T\, y_{k-1}$ is a real number. Thus, implementing these equations is relatively easy and can be accomplished with reasonably small computational resources.

After finding p_1 we then want to calculate the d_1^2 value of the covariance matrix of the source. One easy way to do this is to select, say, 100 randomly chosen images $x_1, x_2, \ldots, x_{100}$ and carry out the following calculation

$$d_1^2 \cong \frac{1}{100}\left| x_1 (x_1^T\, p_1) + x_2 (x_2^T\, p_1) + \cdots + x_{100} (x_{100}^T\, p_1) \right| \cong \left| XX^T\, p_1 \right| \quad .$$

This process gives us p_1 and d_1. To get the remaining useful columns of P and their singular values we confine ourselves to the subspace perpendicular to p_1 and repeat this process to get p_2 and d_2. We then confine ourselves to the subspace perpendicular to both p_1 and p_2 (these vectors should turn out to be approximately orthogonal) and then find p_3 and d_3; and so on, up to p_r and d_r. How we confine ourselves to these subspaces is explained next.

On iteration j of the above process we use the same equations but our initial y vector and all of our x vectors are projected into the subspace of R^m that is perpendicular to $p_1, p_2, \ldots, p_{j-1}$. To take an arbitrary vector z in R^m and project it into a vector $\tilde{z}$ in this subspace, we simply carry out the following calculation (called *Gram-Schmidt orthogonalization*)

$$\tilde{z} = z - (z^T p_1) p_1 - (z^T p_2) p_2 - \cdots - (z^T p_{j-1}) p_{j-1} \quad .$$

Of course, for a real data source, we do not know the value of r (the rank of X). A convenient method is to plot the singular values $d_1, d_2, d_3, \ldots, d_{50}, \ldots$ as they emerge from this process. It is usually very obvious where these become unimportant (although it is not possible to know for sure until an error analysis is done with the reduced-dimensionality representation, as described above).

The main point is that most real-world data sources have an amazingly low effective r value. This fact (and the r principal components) can be used in a variety of ways.

References

1. Diamantaras KI and Kung SY (1996) Principal Component Neural Networks. New York: Wiley.

2. Kohonen T (1989) Self-Organization and Associative Memory, Third Edition. Berlin: Springer.

