

# Learning in Multidimensional Spaces — Neural Networks. Matrix/Tensor Formulation

Andrew P Papliński  
Monash University, Faculty of Information Technology  
Technical Report

May 4, 2018

(This is a work in progress. Expect lots of small errors!)

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	The scope of the report . . . . .	3
1.2	Generic problem formulation . . . . .	3
1.3	Classification of learning problems — modelling the data . . . . .	5
<b>2</b>	<b>Linear mapping</b>	<b>9</b>
2.1	Linear Problem Specification . . . . .	9
2.2	Analytical solution . . . . .	10
2.3	Examples . . . . .	14
<b>3</b>	<b>Fundamentals of Non-Linear mapping and learning algorithms</b>	<b>17</b>
3.1	Gradient descent methods . . . . .	17
3.2	The Newton method . . . . .	19
3.3	The Gauss-Newton and Levenberg-Marquardt algorithms . . . . .	20
<b>4</b>	<b>More on non-linear learning algorithms</b>	<b>22</b>
4.1	Why gradient-decent algorithms are slow . . . . .	22
4.1.1	Examples of error surfaces . . . . .	22
4.1.2	Illustration of sensitivity to a learning rate . . . . .	24
4.2	Heuristic Improvements to the Back-Propagation Algorithm . . . . .	24
4.2.1	The momentum term . . . . .	26
4.2.2	Adaptive learning rate . . . . .	26
4.3	Line search minimisation procedures . . . . .	28
4.4	Conjugate Gradient Algorithms . . . . .	29

4.5	The Adam learning/optimization algorithm . . . . .	31
<b>5</b>	<b>Expectation Maximization Algorithm in Point Set Registration</b>	<b>32</b>
5.1	Point Set Registration Fundamentals . . . . .	32
5.2	Expectation Maximization (EM) Algorithm . . . . .	34
5.3	Affine point set registration . . . . .	36
5.4	Rigid point set registration . . . . .	40
5.5	Nonrigid point set registration — Coherent Point Drift Algorithm . . . . .	42
<b>6</b>	<b>Structure of Neural Networks</b>	<b>44</b>
6.1	Biological Foundations of Neural Networks . . . . .	44
6.2	A simplistic model of a biological neuron . . . . .	46
6.3	Models of artificial neurons . . . . .	48
6.4	Types of activation functions . . . . .	50
6.5	A layer of neurons . . . . .	53
6.6	Feedforward Multilayer aka Deep Neural Networks . . . . .	55
6.7	Two-layer neural network . . . . .	55
6.8	Example of a function implemented by a two-layer nnet . . . . .	58
<b>7</b>	<b>Learning Algorithms for Feedforward/Deep Neural Networks</b>	<b>59</b>
7.1	Fundamentals of Error-Correcting Learning Algorithms . . . . .	59
7.2	Backpropagation of Errors in Two-layer nnets . . . . .	61
7.2.1	The Last (output) Layer . . . . .	61
7.2.2	The Hidden Layer . . . . .	62
7.2.3	Summary of learning in two-layer nnet . . . . .	64
7.3	Pattern and batch learning . . . . .	64
7.4	(Simple) example of function approximation . . . . .	66
7.5	Learning in deep neural networks . . . . .	70
<b>8</b>	<b>Softmax Classifier</b>	<b>71</b>
8.1	A linear Softmax classifier example . . . . .	73
8.2	A two-layer Softmax classifier spiral data example . . . . .	76
8.3	The MNIST example: Hand-written digits classification . . . . .	79
<b>9</b>	<b>Convolutional Neural Networks</b>	<b>81</b>
9.1	Preliminary considerations . . . . .	81
9.2	Convolution fundamentals . . . . .	82
9.2.1	Numerical example . . . . .	85
9.2.2	The proof . . . . .	86
9.3	A basic building block of a convolution Layer . . . . .	87

# 1 Introduction

## 1.1 The scope of the report

This material in this report is considered to be well-known. The original sources can be traced through, e.g., <https://en.wikipedia.org>. If you use this Report, please refer to it as: title, author, Monash University, FIT Technical Report *date*, url.

The material is aimed at graduate students working in problems related to learning. As the foundation concepts we will be operating with points and vectors in multidimensional spaces. For this we will use knowledge from the linear algebra and multivariate calculus.

Learning and mapping sets of points between multidimensional spaces is a common problem considered in many areas, for example:

- Machine learning as in multilayer neural networks, deep learning in particular,
- Multivariate linear and non-linear regression in statistics,
- Linear and non-linear control systems and signal processing,
- Procrustes analysis in statistics and computer vision,
- Point Set Registration
- Active Shape and Active Appearance Models in 2D and 3D computer vision.

The list can be made as long as we wish.

**The background knowledge** for the first part includes linear algebra and multivariable calculus. In particular the essential concepts from linear algebra include: points and related vectors in multidimensional spaces, inner (dot) products of two vectors, matrices. From the calculus we need a concept of a “scalar” function of a vector, partial derivatives and gradients.

## 1.2 Generic problem formulation

We start with formulation of a problem generic, in particular, to supervised learning and equivalent concepts.

**Given data as:**

- $N$  points/vectors  $\mathbf{x}_n$  of dimensionality  $D$
- $N$  points/vectors  $\mathbf{y}_n$  of dimensionality  $D_y$

Data points  $\mathbf{x}_n$  and  $\mathbf{y}_n$  are typically organized in matrices  $X$  and  $Y$ , one vector per column, of dimensions  $D \times N$  and  $D_y \times N$ , respectively:

$$X = \begin{bmatrix} \mathbf{x}_1 & \cdots & \mathbf{x}_n & \cdots & \mathbf{x}_N \end{bmatrix} \quad \text{and} \quad Y = \begin{bmatrix} \mathbf{y}_1 & \cdots & \mathbf{y}_n & \cdots & \mathbf{y}_N \end{bmatrix} \quad (1.1)$$

Note that, by convention, a  $D$ -dimensional vector  $\mathbf{x}$  is represented by a  $D \times 1$  matrix (column vector). Transposing  $\mathbf{x}$ , we get a row-vector  $\mathbf{x}^T$  (a  $1 \times D$  matrix).

**Find:**

- parameters  $\mathbf{w}$  of a non-linear, in general, parametric function:

$$Z = F(X; \mathbf{w}) \quad \text{such that} \quad Z \approx Y \quad (1.2)$$

Note that the size of the matrix  $Z$  is the same as the matrix  $Y$ , namely  $D_y \times N$ . The function  $F(\cdot)$  can be implemented by a neural network as explained in the subsequent sections.

In sec. 5, we will consider a slightly different case in which the numbers of points  $X$  and  $Z$  is different than number of target points  $Y$ . That will make considerations a bit more difficult. For now, we consider the case as specified above, that is, the number of points  $X$ ,  $Z$ , and  $Y$  are the same, and equal to  $N$ .

The function  $F(\cdot)$  is typically interpreted on a point-by-point basis as

$$\mathbf{z}_n = F(\mathbf{x}_n; \mathbf{w}) \quad (1.3)$$

Typically, the points  $\mathbf{z}_n$  approximate the points  $\mathbf{y}_n$  wrt some measure, the simplest being a function of errors  $\mathbf{z}_n - \mathbf{y}_n$ . It means that there is a function, often called a **loss function**, measuring the total approximation error, and the idea is to find the optimal vector of parameters  $\mathbf{w}$  that minimises this function of errors. We can see, that this can be considered as an optimization problem.

More precisely, we can specify:

- The point error:

$$\mathbf{e}_n = \mathbf{z}_n - \mathbf{y}_n \quad (1.4)$$

Note that each point-error  $\mathbf{e}_n$  is an  $D_y$ -dimensional vector.

- In the most common case of the Least Mean Square (LMS) approximation, the total error function  $E(\mathbf{w})$  is the sum of all squared errors:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^N \|\mathbf{e}_n\|^2 = \frac{1}{2} \sum_{n=1}^N \mathbf{e}_n^T \cdot \mathbf{e}_n = \sum_{n=1}^N E_n(\mathbf{w}) ; \quad \text{where} \quad E_n(\mathbf{w}) = \frac{1}{2} \mathbf{e}_n^T \cdot \mathbf{e}_n \quad (1.5)$$

Note that  $E(\mathbf{w})$  and the point error function  $E_n(\mathbf{w})$  are scalar functions of a vector parameter, that is, multivariate or multivariable functions. Note also that we use an **inner product** of the error vector to find the square of its length (norm):

$$\|\mathbf{e}_n\|^2 = \mathbf{e}_n^T \cdot \mathbf{e}_n$$

We could divide  $E(\mathbf{w})$  by the total number of points  $N$  to have a **mean square error**. However, since we are looking for the parameter  $\mathbf{w}$  for which the error function  $E(\mathbf{w})$  attains minimum, multiplication by a constant does not change anything.

Finally, it is not uncommon to weight individual errors with known parameters, using  $M \times M$  fixed matrix  $Q$  in the following way:

$$E(\mathbf{w}) = \sum_{n=1}^N E_n(\mathbf{w}) ; \text{ where } E_n(\mathbf{w}) = \frac{1}{2} \mathbf{e}_n^T \cdot Q \cdot \mathbf{e}_n \quad (1.6)$$

### 1.3 Classification of learning problems — modelling the data

[You can skip this section in your first reading]

Given the data points we try to model them looking at the following generic possibilities:

#### Supervised Learning:

Given a set of data points as in eqn.(1.1) find an approximating parameterized function as in eqn.(1.2). Fig. 1–1 shows collection of 1-D points ,  $\{x, y\}$  and  $z = F(x)$ .

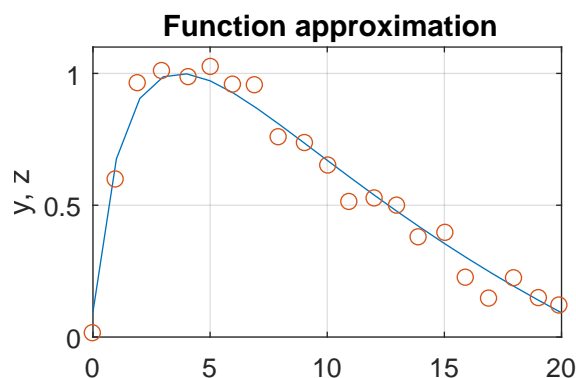


Figure 1–1: The data points  $y$  (circles) are approximated by a function  $z = F(x)$  (continuous line)

In a more general way, let us consider the following mapping,  $f(X)$ , from a  $D$ -dimensional domain  $X$  into an  $M$ -dimensional output space  $Y$  as in Fig. 1–2

A function:

$$f : X \rightarrow Y , \quad \text{or} \quad \mathbf{y} = f(\mathbf{x}) ; \quad \mathbf{x} \in X \subset \mathcal{R}^D , \quad \mathbf{y} \in Y \subset \mathcal{R}^M$$

is assumed to be unknown, but it is specified by a set of learning examples,  $\{X; Y\}$ , that is, the set of expected input-output pairs.

In particular, in classification problem, the output signals,  $\mathbf{y}$ , represent labels of the classes the input signals  $\mathbf{x}$  belong to.

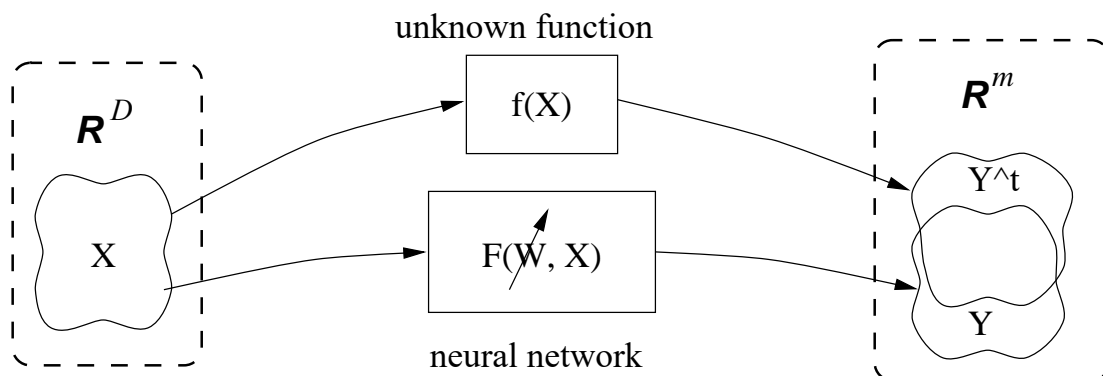


Figure 1–2: Mapping from a  $D$ -dimensional domain into an  $M$ -dimensional output space

The unknown function  $f$  is approximated by a fixed, parameterised function  $F$ :

$$F : \mathcal{R}^D \times \mathcal{R}^M \rightarrow \mathcal{R}^m, \quad \text{or} \quad \mathbf{z} = F(W, \mathbf{x}); \quad \mathbf{x} \in \mathcal{R}^D, \quad \mathbf{z} \in \mathcal{R}^m, \quad W \in \mathcal{R}^M$$

where  $W$  here is a total set of  $M$  parameters, arranged into appropriate weight matrices.

The function  $F$  is often implemented as a multi-layer neural network that we will discuss in the subsequent sections.

Approximation of the unknown function  $f$  is performed in such a way that during the **supervised learning procedure** some **performance index**, or the **loss function**  $L$ , a function of the weight parameters  $W$ , the set of learning pairs  $X$  and  $Z$ ,

$$L = L(X, Y, Z; W)$$

is minimised. A good candidate for a loss function is the error function of eqns (1.5), (1.6).

We will discuss details of the learning procedure in subsequent sections.

## Unsupervised Clustering

In this case, the assumption is that data is organized in clusters of points in  $D$ -dimensional space.

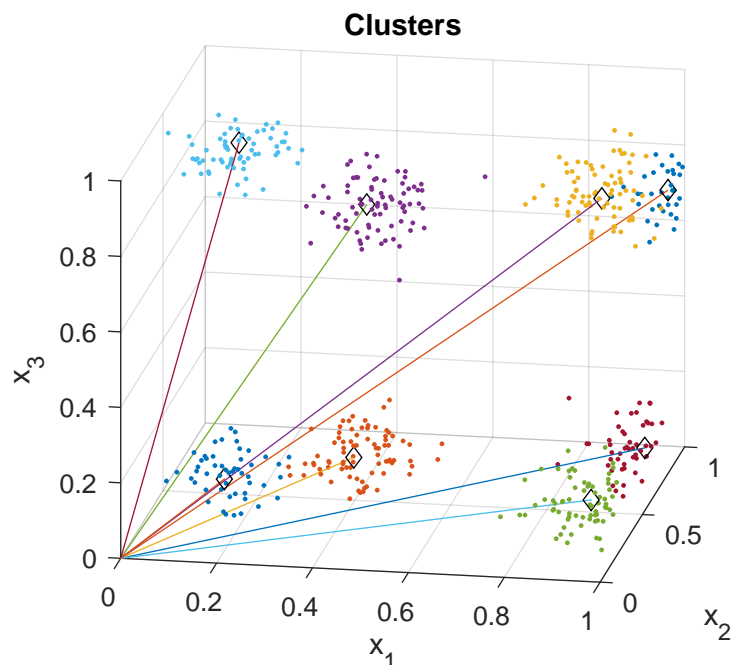


Figure 1–3: Example of  $m = 8$  clusters of datapoints in a  $D = 3$ -dimensional space

In the example we have  $M$  clusters and some number of points in each cluster. The dimensionality of the data space in the example is  $D = 3$ . The unsupervised learning algorithm should discover the centroids/means of the clusters marked in Fig. 1–3 with diamond shapes. There are a number of clustering algorithms, the **k-means algorithm** ( $k = M$ ) being the most widely used.

The MATLAB code that generates Fig. 1–3 can be found in <http://users.monash.edu/~app/Lrn/Mt1b>

## Supervised Clustering – Classification

In the case of classification, the clusters of data points have labels assigned to each cluster. We can have clusters of images of cats, dogs, or any other objects. The algorithm, or related neural network would have an output for each category of objects. Each output should produce a probability of a given data point to belong to a specific class. In sec. 8 we consider a popular **Softmax Classifier**.

### Principal components analysis

Conceptually, the objective of the Principal Components Analysis is to analyze a shape of a single-cluster data. The datapoints are encapsulated inside a hyper-ellipsoid and we need to find out the directions, and lengths of all axes.

(from [https://en.wikipedia.org/wiki/Principal\\_component\\_analysis](https://en.wikipedia.org/wiki/Principal_component_analysis))

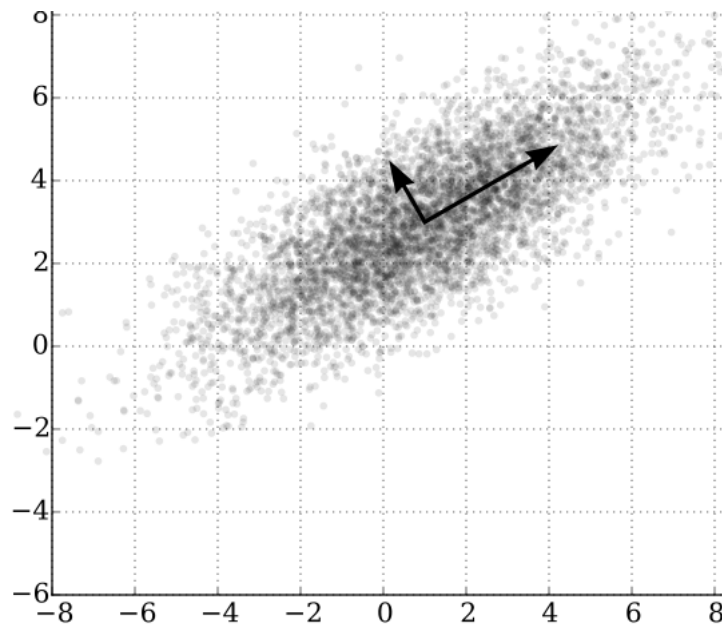


Figure 1–4: Example of PCA in 2D

The example in Fig. 1–4 shows an ellipsoidal cloud of 2D points with the principal vectors. The MATLAB code for a 3D case can be found in <http://users.monash.edu/~app/Lrn/Mtlb>. The related neural network that discovers the principal components of the datapoint implements the generalized Hebbian learning.



## 2 Linear mapping

At this point it is a good idea to make sure that you are familiar with the notation introduced in sec. 1.2.

### 2.1 Linear Problem Specification

As a special case of eqn (1.2) we consider a linear mapping that can be defined in the following way:

$$\mathbf{z}_n = A \cdot \mathbf{x}_n + \mathbf{b} = W \cdot \hat{\mathbf{x}}_n ; \text{ where } W = [A \ \mathbf{b}] \text{ and } \hat{\mathbf{x}}_n = \begin{bmatrix} \mathbf{x}_n \\ 1 \end{bmatrix} \quad (2.1)$$

The matrix of the unknown parameters  $W$  is composed of a  $D_y \times D$  matrix  $A$  and an  $M$ -dimensional vector  $\mathbf{b}$  referred to as a bias vector, so that the dimension of the matrix  $W$  is  $D_y \times (D + 1)$ . The vector  $\mathbf{x}_n$  is prepended with 1 to form an extended vector  $\hat{\mathbf{x}}_n$ . In specific applications eqn (2.1) can define, among many others:

- an affine transform,
- a rigid body transform. In this case  $\mathbf{b}$  is the translation vector, and the matrix  $A = s \cdot R$  includes the scaling factor  $s$  and the unitary rotation matrix such that  $R^T R = I$  and  $\det R = 1$ .
- a linear part of a neural network layer. In this case  $W$  is a weight matrix and  $\mathbf{b}$  part is the bias. We will consider this case in greater detail in the subsequent sections.

The extended, or augmented vector  $\hat{\mathbf{x}}_n$  is typically used in the homogenous coordinates and its usage simplify the notation by incorporating the bias vector into a single the matrix of parameters. Using this notation, eqn (2.1) can be generalized for all  $N$  points as:

$$Z = W \cdot \hat{X} ; \text{ where } \hat{X} = \begin{bmatrix} X \\ \mathbf{1}_N^T \end{bmatrix} \quad (2.2)$$

where  $\mathbf{1}_N$  is a vector of  $N$  ones. Note that we multiply the  $D_y \times (D + 1)$ -dimensional matrix of parameters  $W$  by the  $(D + 1) \times N$ -dimensional matrix of extended points  $\hat{X}$ , which results in the  $D_y \times N$ -dimensional matrix of transformed points  $Z$ .

The point error function specified in eqn (1.5) can be now calculated as

$$\begin{aligned} 2E_n(W) &= \mathbf{e}_n^T \cdot \mathbf{e}_n = (\mathbf{z}_n - \mathbf{y}_n)^T \cdot (\mathbf{z}_n - \mathbf{y}_n) \\ &= (W \cdot \hat{\mathbf{x}}_n - \mathbf{y}_n)^T \cdot (W \cdot \hat{\mathbf{x}}_n - \mathbf{y}_n) \end{aligned}$$

Multiplying it out gives

$$E_n(W) = \frac{1}{2} (\hat{\mathbf{x}}_n^T \cdot W^T \cdot W \cdot \hat{\mathbf{x}}_n - 2\mathbf{y}_n^T \cdot W \cdot \hat{\mathbf{x}}_n + \mathbf{y}_n^T \cdot \mathbf{y}_n) \quad (2.3)$$

Note that the point error function  $E_n(W)$  is a quadratic **scalar function** of the matrix  $W$ , that is a quadratic scalar function of all elements of the matrix  $W$ . The total error function  $E(W)$  is a sum of all point error functions as in eqn (1.5). In the space of the weight parameter,  $E(W)$  is a hyper-paraboloid.

## 2.2 Analytical solution

Before we proceed make sure that you are confident with the vector/matrix differentiation. As a background reading you might like to refer to [https://en.wikipedia.org/wiki/Matrix\\_calculus](https://en.wikipedia.org/wiki/Matrix_calculus), or any alternative source of information.

The plan is to form the quadratic error function  $E(W)$  of parameters  $W$  as in eqns (2.3) and (1.5), calculate the derivative of  $E(W)$  wrt  $W$  and find the value of the parameters  $W$  for which the derivative attains 0. For such a matrix  $W$  the error function  $E(W)$  attains minimum.

We start with the following expression for the derivative:

$$\frac{\partial E(W)}{\partial W} = \sum_{n=1}^N \frac{\partial E_n(W)}{\partial W} \quad (2.4)$$

Trying to calculate  $\frac{\partial E_n(W)}{\partial W}$ , where  $E_n(W)$  is specified in eqn (2.3), will unavoidably results in multidimensional objects known as tensors, a matrix being a tensor of the order 2.

In order to avoid tensors, that is, multidimensional collections of numbers, and operate with vectors and matrices only, it is convenient to have an operation of converting a matrix into a vector. This is done by stacking the columns of a matrix into one “long” vector, so that we can write:

$$\mathbf{w} = \text{vec}(W) = \downarrow W \quad (2.5)$$

In general, if  $W$  is an  $r \times c$  matrix, then  $\mathbf{w}$  is an  $(r \times c)$ -dimensional vector. If you use MATLAB, then the column-stacking operation is performed by  $\mathbf{w} = W(:)$ .

The column-stacking of the  $D_y \times (D + 1)$  matrix  $W$  results in the vector  $\mathbf{w}$  of dimensionality:

$$D_w = D_y \times (D + 1)$$

The error function  $E(W)$  will become  $E(\mathbf{w})$ , that is, a scalar function of a vector argument and the derivative of eqn (2.4) becomes the gradient of the error function wrt  $\mathbf{w}$ :

$$\nabla_{\mathbf{w}} E(\mathbf{w}) = \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = \sum_{n=1}^N \frac{\partial E_n(\mathbf{w})}{\partial \mathbf{w}} \quad (2.6)$$

Eqn (2.6) states that the total gradient  $\nabla_{\mathbf{w}} E(\mathbf{w})$  is a  $D_w$ -dimensional vector and that it can be calculated as a sum of the point gradients  $\frac{\partial E_n(\mathbf{w})}{\partial \mathbf{w}}$ . Recalling the rule of the derivative of a quadratic function, the chain rule, and using eqns (1.5) and (1.4), we have

$$\frac{\partial E_n(\mathbf{w})}{\partial \mathbf{w}} = \frac{\partial \mathbf{e}_n^T(\mathbf{w})}{\partial \mathbf{w}} \cdot \mathbf{e}_n = \frac{\partial \mathbf{z}_n^T(\mathbf{w})}{\partial \mathbf{w}} \cdot \mathbf{e}_n = \mathcal{J}_n(\mathbf{z}_n, \mathbf{w}) \cdot \mathbf{e}_n = \mathcal{J}_n(\mathbf{z}_n, \mathbf{w}) \cdot (\mathbf{z}_n - \mathbf{y}_n) \quad (2.7)$$

where

$$\mathcal{J}(\mathbf{z}, \mathbf{w}) = \frac{\partial \mathbf{z}^T(\mathbf{w})}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial z_1}{\partial w_1} & \dots & \frac{\partial z_M}{\partial w_1} \\ \vdots & \dots & \vdots \\ \frac{\partial z_1}{\partial w_{D_w}} & \dots & \frac{\partial z_M}{\partial w_{D_w}} \end{bmatrix} \quad (2.8)$$

is the  $D_w \times D_y$  Jacobian matrix of the first derivatives of  $\mathbf{z}_n$  wrt the vector of parameters  $\mathbf{w}$ . Note that for notational convenience, in eqn (2.8) the subscript  $n$  has been dropped. Since a Jacobian is a derivative of a vector with respect to another vector, it is important to agree on the order of writing derivatives. In our case the rows of the Jacobian follow the transposed vector  $\mathbf{z}^T$ , whereas the columns are arranged in the order of the denominator vector  $\mathbf{w}$ .

Combining eqns (2.6) and (2.7), and substituting eqn (2.1), the total gradient can be expressed as:

$$\nabla_{\mathbf{w}} E(\mathbf{w}) = \sum_{n=1}^N \mathcal{J}_n(\mathbf{z}_n, \mathbf{w}) \cdot (W \cdot \hat{\mathbf{x}}_n - \mathbf{y}_n) \quad (2.9)$$

Equating the gradient to zero, gives the equation for the optimal matrix of parameters  $W$  in the form

$$\sum_{n=1}^N \mathcal{J}_n(\mathbf{z}_n, \mathbf{w}) \cdot W \cdot \hat{\mathbf{x}}_n = \sum_{n=1}^N \mathcal{J}_n(\mathbf{z}_n, \mathbf{w}) \cdot \mathbf{y}_n \quad (2.10)$$

Note that temporarily, the parameters are in two forms: as a matrix  $W$  and the column-stacked vector  $\mathbf{w}$ . Since the gradient is  $D_w$ -dimensional, we can think about eqn (2.10) as a set of  $D_w = D_y \times (D + 1)$  scalar linear equations for the unknown parameters  $W$ .

The next task is to calculate the Jacobian matrix needed in eqn (2.10). One way of doing so is to represent the matrix of parameters  $W$  as a collection of column vectors

$$W = \begin{bmatrix} \mathbf{w}_1 & \mathbf{w}_2 & \dots & \mathbf{w}_{D+1} \end{bmatrix}$$

so that the vector  $\mathbf{w}$  stacks all column vectors  $\mathbf{w}_i$ . We can now calculate  $\mathbf{z}_n$  of eqn (2.1) as a block inner product as follows

$$\mathbf{z}_n = W \cdot \hat{\mathbf{x}}_n = \sum_{i=1}^{D+1} \mathbf{w}_i \cdot x_{in} \quad (2.11)$$

where  $x_{in}$  is the  $i$ -th component of the  $n$ -th point. Referring to eqn (2.2) note that all  $x_{D+1n} = 1$ . Now, to find the Jacobian as in eqn (2.8), we differentiate  $\mathbf{z}_n$  of eqn (2.11) with respect to all  $\mathbf{w}_i$ . For each  $i$  we calculate a block of the Jacobian matrix in the following way:

$$\frac{\partial \mathbf{z}_n^T}{\partial \mathbf{w}_i} = x_{in} \cdot I_{D_y} \quad \text{for } i = 1 \dots D + 1$$

where  $I_{D_y}$  is the  $D_y \times D_y$  identity matrix. Arranging the above derivatives into a complete **point Jacobian matrix** we have:

$$\mathcal{J}_n(\mathbf{z}_n, \mathbf{w}) = \frac{\partial \mathbf{z}_n^T}{\partial \mathbf{w}} = \begin{bmatrix} x_{1n} \cdot I_{D_y} \\ x_{2n} \cdot I_{D_y} \\ \vdots \\ x_{D_n n} \cdot I_{D_y} \\ I_{D_y} \end{bmatrix} = \hat{\mathbf{x}}_n \otimes I_{D_y} \quad (2.12)$$

where ‘ $\otimes$ ’ denotes the Kronecker product. This is a really simple and elegant result. Each  $D_y \times D_y$  block of the Jacobian, is a diagonal matrix containing a single component of the  $n$ -th input vector, i.e.,  $x_{in} \cdot I_{D_y}$ . The important aspect to notice is that the Jacobian is independent of the parameters  $\mathbf{w} = \downarrow W$ . This is not a surprise since the  $\mathbf{z}_n$  as in eqn (2.11) is a linear function of the parameters  $W$  and the Jacobian consists of first derivatives of the parameters.

In order to transform eqn (2.10) for the unknown parameters  $W$  into a more efficient form, we have to calculate a product of a Jacobian and a vector. We start with the RHS of eqn (2.10) and use the result (2.12):

$$\mathcal{J}_n(\mathbf{z}_n, \mathbf{w}) \cdot \mathbf{y}_n = (\hat{\mathbf{x}}_n \otimes I_{D_y}) \cdot \mathbf{y}_n \quad (2.13)$$

Now we can use the following well known identity related to mixing ordinary matrix products with the Kronecker products. It states that for appropriately sized matrices  $A, B, C, D$ , we have:

$$(A \otimes B) \cdot (C \otimes D) = (A \cdot B) \otimes (C \cdot D) \quad (2.14)$$

Using (2.14) we can transform (2.13) further as:

$$(\hat{\mathbf{x}}_n \otimes I_{D_y}) \cdot \mathbf{y}_n = \hat{\mathbf{x}}_n \otimes \mathbf{y}_n \quad (2.15)$$

We can now use another mathematical identity that states that a Kronecker product of two vectors  $\mathbf{a}, \mathbf{b}$  of any dimensions can be replaced by a vectorised outer product of those vectors, namely:

$$\mathbf{a} \otimes \mathbf{b} = \downarrow(\mathbf{b} \cdot \mathbf{a}^T) \quad (2.16)$$

Note the the outer product  $\mathbf{b} \cdot \mathbf{a}^T$  is a  $D_b \times D_a$  matrix (of rank one), unlike the inner product  $\mathbf{b}^T \cdot \mathbf{a}$  which is a scalar. Note also the change of order of  $\mathbf{a}, \mathbf{b}$ . Utilizing the identities (2.15) and (2.16), we can further write eqn (2.13), which is the RHS of eqn (2.10), in the following simple form:

$$\mathcal{J}_n(\mathbf{z}_n, \mathbf{w}) \cdot \mathbf{y}_n = \frac{\partial \mathbf{z}_n^T}{\partial \mathbf{w}} \cdot \mathbf{y}_n = \downarrow(\mathbf{y}_n \cdot \hat{\mathbf{x}}_n^T) \quad (2.17)$$

This is an important result that states that in the linear case, when  $\mathbf{z}_n = W \cdot \hat{\mathbf{x}}_n$  the product of the Jacobian matrix and a vector  $\mathbf{y}_n$  is equal to the vectorised **outer product** of  $\mathbf{y}_n$  and  $\hat{\mathbf{x}}_n$ . We will use this results in discussing neural networks

Similarly, the LHS of eqn (2.10) can be expressed as

$$\mathcal{J}_n(\mathbf{z}_n, \mathbf{w}) \cdot W \cdot \hat{\mathbf{x}}_n = \downarrow(W \cdot \hat{\mathbf{x}}_n \cdot \hat{\mathbf{x}}_n^T) \quad (2.18)$$

Combining (2.18) and (2.17) we can write eqn (2.10) for the unknown parameters  $W$  in the following form:

$$\sum_{n=1}^N \downarrow(W \cdot \hat{\mathbf{x}}_n \cdot \hat{\mathbf{x}}_n^T) = \sum_{n=1}^N \downarrow(\mathbf{y}_n \cdot \hat{\mathbf{x}}_n^T) \quad (2.19)$$

Finally after dropping vectorizing of matrices and dividing by  $N$ , we arrive at the following simple equation for the unknown matrix of parameters  $W$ :

$$W \cdot \hat{R} = \hat{C} \quad \text{hence} \quad W = \hat{C} \cdot \hat{R}^{-1} \quad (2.20)$$

where

$$\hat{R} = \frac{1}{N} \sum_{n=1}^N \hat{\mathbf{x}}_n \cdot \hat{\mathbf{x}}_n^T = \frac{1}{N} \hat{X} \cdot \hat{X}^T \quad (2.21)$$

is the (extended) input **auto-correlation matrix** of size  $(D+1) \times (D+1)$ , and

$$\hat{C} = \frac{1}{N} \sum_{n=1}^N \mathbf{y}_n \cdot \hat{\mathbf{x}}_n^T = \frac{1}{N} Y \cdot \hat{X}^T \quad (2.22)$$

is the **cross-correlation matrix** of size  $D_y \times (D+1)$ .

(Before we move further we might like to examine the difference between the correlation and covariance matrices. We need to re-check which values we use here.)

Eqn (2.20) is one of the most famous results in the LMS optimization. In equivalent versions it is known as the **normal equation**, the **Wiener-Hopf equation** and as a number of other names used over its long history.

Note that:

- Eqn (2.20) is a linear matrix equation with the unknown matrix of parameters  $W$ . It makes sense, since the original mapping as in eqn (2.1) is linear.
- The auto-correlation matrix  $\hat{R}$  is a symmetrical, positive definite matrix. This property flows from the fact that the main diagonal consists of the squares of all  $x_{in}$ . If  $N > D$  and  $\mathbf{x}_n$  vectors are linearly independent, it will be a full rank matrix, i.e., invertible.

The extended auto-correlation matrix  $\hat{R}$  of eqn (2.21) can be further expressed as:

$$\hat{R} = \frac{1}{N} \begin{bmatrix} X \\ \mathbf{1}_N^T \end{bmatrix} \cdot \begin{bmatrix} X^T & \mathbf{1}_N \end{bmatrix} = \frac{1}{N} \begin{bmatrix} X \cdot X^T & X \cdot \mathbf{1}_N \\ \mathbf{1}_N^T \cdot X^T & \mathbf{1}_N^T \cdot \mathbf{1}_N \end{bmatrix} = \begin{bmatrix} R & \bar{\mathbf{x}} \\ \bar{\mathbf{x}}^T & 1 \end{bmatrix} \quad (2.23)$$

where

$$R = \frac{1}{N} X \cdot X^T ; \quad \bar{\mathbf{x}} = \frac{1}{N} X \cdot \mathbf{1}_N = \frac{1}{N} \sum_{n=1}^N \mathbf{x}_n \quad (2.24)$$

are the “proper”  $D \times D$  auto-correlation matrix  $R$ , and the mean input vector  $\bar{\mathbf{x}}$ , respectively. Similarly, the extended cross-correlation matrix (2.22) can be further expressed as:

$$\hat{C} = \frac{1}{N} Y \cdot \begin{bmatrix} X^T & \mathbf{1}_N \end{bmatrix} = \frac{1}{N} \begin{bmatrix} Y \cdot X^T & Y \cdot \mathbf{1}_N \end{bmatrix} = \begin{bmatrix} C & \bar{\mathbf{y}} \end{bmatrix} \quad (2.25)$$

where

$$C = \frac{1}{N} Y \cdot X^T ; \quad \bar{\mathbf{y}} = \frac{1}{N} Y \cdot \mathbf{1}_N = \frac{1}{N} \sum_{n=1}^N \mathbf{y}_n \quad (2.26)$$

are the “proper”  $D_y \times D$  cross-correlation matrix  $C$ , and the mean desired vector  $\bar{\mathbf{y}}$ , respectively.

## 2.3 Examples

Let us consider a rather demanding case of mapping 3D points located on some surface on a 2D shape, as illustrated in Fig. 2–1.

**A 3-D shape to be mapped onto a 2-D shape**

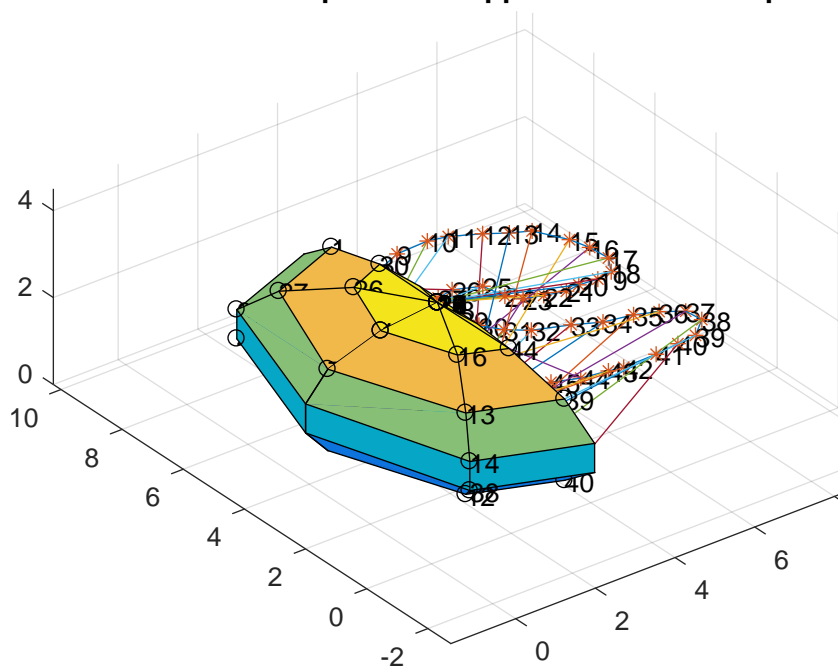


Figure 2–1: A set of 3D points to be mapped onto a set of 2D points

The MATLAB code can be found in <http://users.monash.edu/~app/Lrn/Mtlb/Lin1.m>  
There are  $N = 50$  points and the first 16 points  $X$  are as follows

```
X(:,1:16) =
2.0  0.1  0.9 -0.4  0.8 -0.4 -1.1  0.3  0.3  0.3  0.3  0.1  0.1 -0.4  0.8  0.9
5.8  5.1  3.7  5.8  4.1  5.8  2.0  2.0  2.0  2.0  2.0 -1.1 -1.1 -1.8 -0.1  0.3
3.9  2.1  1.6  2.7  2.7  3.3  3.9  1.6  1.6  4.4  4.4  2.1  3.9  3.3  3.3  4.4
```

The points  $X$  are located on a 3D surface. The first 16 2D target points  $Y$  are as follows:

```
Y(:,1:16) =
3.8  3.9  4.3  4.8  5.3  5.5  5.9  6.2  7.0  7.8  8.2  8.8  9.3  9.7  10.0  10.2
6.9  7.2  7.7  8.4  8.6  8.9  9.5  9.9  10.2  10.3  10.1  9.8  9.5  9.3  8.6  8.1
```

Now, we are going to perform the linear mapping as in eqn (2.2), and the problem is to find the matrix  $W$  that makes the points  $Z$  as close as possible to  $Y$ . We start with eqn (2.23) to calculate the extended autocorrelation matrix  $\hat{R}$ :

```
wNs = ones(1,N) ;
Xh = [X ; wNs] ;
Rh = Xh*Xh'/N ;
```

to get:

```
Rh = 1.86  2.14  3.20  1.09
      2.14  8.19  5.95  1.98
      3.20  5.95  10.39  2.99
      1.09  1.98  2.99  1.00
```

You can compare this result with the structure presented in eqn (2.23). Subsequently, we calculate the extended cross-correlation matrix  $\hat{C}$  as in eqn (2.25)

```
Ch = Y*Xh'/N ;
```

to get:

```
Ch = 8.12  11.70  22.99  7.43
      6.60  14.80  21.59  6.95
```

Finally, we can calculate the mapping matrix of parameters  $W$  as in eqn (2.20):

```
W = Ch/Rh % left division is used instead of an inverse
    = 0.064 -0.711  0.542  7.149
      -1.392  0.235  0.492  6.529
```

Finally we can map points  $X$  into points  $Z$  and compare with points  $Y$ :

```
Z = W*Xh;
mse = sum(sum((Z-Y).^2))/N ; % the mean square error
```

The results of mapping are illustrated in Fig. 2-2. The results of mapping do not look very encouraging, with the  $mse = 2.78$ . This is an early warning that the linear mapping has an obvious limitations. The linear operations involved translation, rotation, scaling and projection. It is rather limited set to mould a 3D shape into a 2D shape accurately.





### 3 Fundamentals of Non-Linear mapping and learning algorithms

Optimization or minimisation of a function of many variables (multi-variable function),  $L(\mathbf{w})$ , has been researched since the XVII century and its principles were formulated by such mathematicians as Kepler, Fermat, Newton, Leibnitz, Gauss.

We start again with the problem formulation as in section 1.2. Go back and inspect eqns (1.1) to (1.5). This time, since the function

$$\mathbf{z}_n = F(\mathbf{x}_n; \mathbf{w})$$

is nonlinear in parameters  $\mathbf{w}$ , the analytical solution to find an optimal vector of parameters that minimises the error function  $E(\mathbf{w})$  of eqn (1.5) most likely does not exist, since each individual point error contains the nonlinearity:

$$\mathbf{e}_n = F(\mathbf{x}_n; \mathbf{w}) - \mathbf{x}_n \quad (3.1)$$

Therefore, the idea is to arrive at the optimal  $\mathbf{w}$  iteratively. We start with an initial value of  $\mathbf{w}$  and then, at each step we calculate an increment  $\Delta\mathbf{w}$  such that

$$E(\mathbf{w} + \Delta\mathbf{w}) < E(\mathbf{w}) \quad ; \quad \mathbf{w} \leftarrow \mathbf{w} + \Delta\mathbf{w} \quad (3.2)$$

The iteration is repeated until the error function is satisfactory small.

In order to find a “good” value of the parameter increment  $\Delta\mathbf{w}$ , we expand  $E(\mathbf{w} + \Delta\mathbf{w})$  into the Taylor power series:

$$E(\mathbf{w} + \Delta\mathbf{w}) = E(\mathbf{w}) + (\nabla E(\mathbf{w}))^T \cdot \Delta\mathbf{w} + \mathcal{O}(\|\Delta\mathbf{w}\|^2) \quad (3.3)$$

where the gradient  $\nabla E(\mathbf{w})$  can be calculated as a sum of point gradients  $\nabla E_n(\mathbf{w})$  as in eqns (2.6) and (2.7). For convenience we re-state that:

$$\nabla E_n(\mathbf{w}) = \frac{\partial \mathbf{e}_n^T(\mathbf{w})}{\partial \mathbf{w}} \cdot \mathbf{e}_n = \mathcal{J}_n(\mathbf{e}_n, \mathbf{w}) \cdot \mathbf{e}_n, \quad \text{where} \quad \mathcal{J}_n = \frac{\partial \mathbf{e}_n^T}{\partial \mathbf{w}} = \frac{\partial \mathbf{z}_n^T}{\partial \mathbf{w}} = \mathcal{J}_n(\mathbf{z}_n, \mathbf{w}) \quad (3.4)$$

For the future reference let us also re-state that the error function  $E(\mathbf{w})$  and its point components  $E_n(\mathbf{w})$  are scalar functions of a vector argument,  $\mathbf{w}$ . The vector  $\mathbf{w}$  and its increment  $\Delta\mathbf{w}$  are  $D_w$ -dimensional vectors. A scalar function of a vector argument is also called a multivariable (or multivariate) function.

#### 3.1 Gradient descent methods

The simplest non-linear optimization method known as the **steepest or gradient descent** is based on the observation that in order to reduce the error as in eqn (3.2), the inner product of the gradient and the increment vector in eqn (3.3) must be negative:

$$(\nabla E)^T \cdot \Delta \mathbf{w} < 0$$

that is, the two vectors should point in the opposite direction, or more precisely, the angle between  $(\nabla E)$  and  $\Delta \mathbf{w}$  should be larger than  $90^\circ$ . Hence, in the steepest decent method of selecting the parameter increment vector we have:

$$\Delta \mathbf{w} = -\eta \nabla E(\mathbf{w}) \quad (3.5)$$

where  $\eta$  is a small positive constant that controls the rate of descent.

Demonstration of the gradient decent method in the simple case of one only scalar parameter  $w$  is shown in Fig. 3-1. In the equivalent linear case  $E(w)$  would be a parabola. The shape of

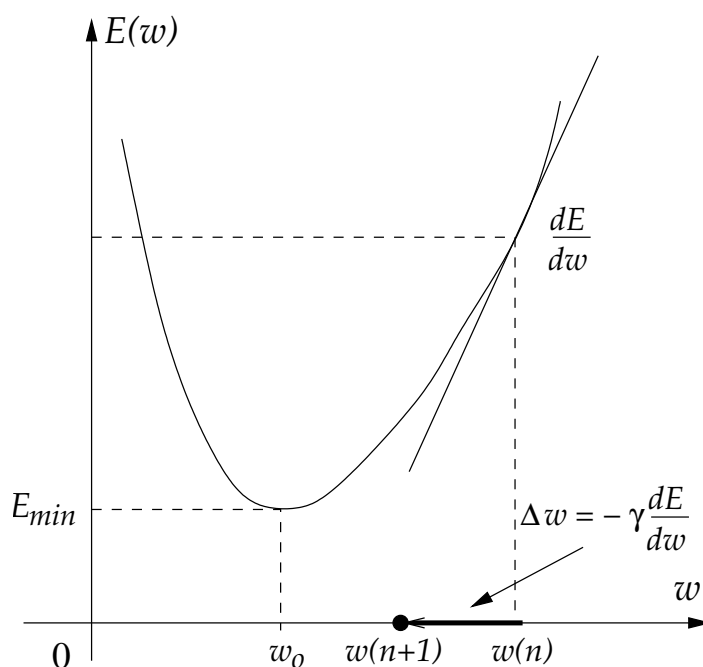


Figure 3-1: Demonstration of a gradient decent concept in the single  $w$  case

$E(w)$  in Fig. 3-1 is similar in that that it has one only minimum for which  $\frac{dE}{dw} = 0$ . We can see that in order to find the optimal value of  $w_o$  for which  $E(w_o)$  attains minimum we modify  $w(n+1) = w(n) + \Delta w$ , where  $\Delta w = -\eta \frac{dE}{dw}$ , as in eqn (3.5).

Note that in multidimensional nonlinear case the error function creates a complex landscape with potentially many local minima for which  $\nabla E(\mathbf{w}) = 0$ . This is a nontrivial problem without a single simple solution.

Using the gradient decent, optimization rule results in a rather slow optimization procedure, therefore, a number of improvements have been developed, some of which are presented in the subsequent sections.

### 3.2 The Newton method

The gradient descent methods use only information from the first derivatives, i.e., the gradient. Using second derivatives will take better into account the shape of the error function, hence, possibly speed up the iteration process.

We calculate the gradient of the Taylor expansion of eqn (3.3) neglecting the higher order terms  $\mathcal{O}(\|\Delta\mathbf{w}\|^2)$ . This gives:

$$\nabla E(\mathbf{w} + \Delta\mathbf{w}) = \nabla E(\mathbf{w}) + \nabla^2 E(\mathbf{w}) \cdot \Delta\mathbf{w} \quad (3.6)$$

where  $\nabla^2 E(\mathbf{w})$  is the  $D_w \times D_w$  matrix of second derivatives known as the **Hessian matrix**:

$$\nabla^2 E(\mathbf{w}) = \mathcal{H}_{E(\mathbf{w})} = \frac{\partial^2 E}{\partial w_i \partial w_j} \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \cdots & \frac{\partial^2 E}{\partial w_1 \partial w_{D_w}} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 E}{\partial w_{D_w} \partial w_1} & \cdots & \frac{\partial^2 E}{\partial w_{D_w}^2} \end{bmatrix} = \frac{\partial \nabla^T E(\mathbf{w})}{\partial \mathbf{w}} \quad (3.7)$$

Note that the Hessian matrix can be expressed as a derivative of the (transposed) gradient wrt the parameter vector  $\mathbf{w}$ .

Since the second partial derivative is independent of the order of differentiation:

$$\frac{\partial^2 E}{\partial w_i \partial w_j} = \frac{\partial^2 E}{\partial w_j \partial w_i}$$

The Hessian matrix is symmetrical:  $(\mathcal{H}_{E(\mathbf{w})})^T = \mathcal{H}_{E(\mathbf{w})}$

In order to find the minimum of the error function we equate the gradient  $\nabla E(\mathbf{w} + \Delta\mathbf{w})$  of eqn (3.6) to zero. It gives the following equation for the increment  $\Delta\mathbf{w}$ :

$$\nabla E(\mathbf{w}) + \mathcal{H}_{E(\mathbf{w})} \cdot \Delta\mathbf{w} = 0$$

Hence, the resulting Newton method states that:

$$\Delta\mathbf{w} = -(\mathcal{H}_{E(\mathbf{w})})^{-1} \cdot \nabla E(\mathbf{w}) \quad (3.8)$$

Comparing with the gradient descent rule (3.5), this time the opposite-to-gradient direction is modified by the inverse of the Hessian matrix.

### 3.3 The Gauss-Newton and Levenberg-Marquardt algorithms

In these two algorithms, the Hessian matrix is approximated by the Jacobian matrix. Note that the total Hessian matrix can be calculated as a sum of the point Hessian matrices as follows:

$$\mathcal{H}_E = \sum_{n=1}^N \mathcal{H}_{E_n}$$

where

$$\mathcal{H}_{E_n} = \frac{\partial \nabla^T E_n}{\partial \mathbf{w}} = \frac{\partial}{\partial \mathbf{w}} (\mathcal{J}_n \cdot \mathbf{e}_n)^T = \frac{\partial}{\partial \mathbf{w}} (\mathbf{e}_n^T \cdot \frac{\partial \mathbf{e}_n}{\partial \mathbf{w}^T}) = \frac{\partial \mathbf{e}_n^T}{\partial \mathbf{w}} \cdot \frac{\partial \mathbf{e}_n}{\partial \mathbf{w}^T} + \mathbf{e}_n^T \cdot \frac{\partial^2 \mathbf{e}_n}{\partial \mathbf{w}^2}$$

Now, we have:

$$\mathcal{H}_{E_n} = \mathcal{J}_n \cdot \mathcal{J}_n^T + \mathbf{e}_n^T \cdot \mathcal{H}_{e_n} \approx \mathcal{J}_n \cdot \mathcal{J}_n^T \quad (3.9)$$

assuming that the errors  $\|\mathbf{e}_n\|$  are small.

Using the above approximation of the Hessian matrix we arrive at a **Gauss-Newton iterative method**:

$$\Delta \mathbf{w} = -\left(\sum_{n=1}^N \mathcal{J}_n \cdot \mathcal{J}_n^T\right)^{-1} \cdot \sum_{n=1}^N \mathcal{J}_n \cdot \mathbf{e}_n \quad (3.10)$$

It is possible to replace summations in eqn (3.10) by arranging all point Jacobians in a block Jacobian matrix of dimension  $D_w \times M \cdot N$  as follows:

$$\mathcal{J}_\epsilon = [\mathcal{J}_1 \dots \mathcal{J}_N] \quad (3.11)$$

Then the sum of products of Jacobians as in eqn (3.10) can be calculated as a product of the block-Jacobians, namely:

$$\sum_{n=1}^N \mathcal{J}_n \cdot \mathcal{J}_n^T = \mathcal{J}_\epsilon \cdot \mathcal{J}_\epsilon^T \quad (3.12)$$

Note that the dimension of the product is  $D_w \times D_w$ .

Similarly, to deal with the second summation in eqn (3.10), we start with stacking all  $N$  error vectors  $\mathbf{e}_n$  in one  $N \cdot M$  vector  $\epsilon$ :

$$\epsilon = \downarrow(Z - Y) = \downarrow[\mathbf{e}_1 \dots \mathbf{e}_N] \quad (3.13)$$

then the sum of products of Jacobians and error vectors of eqn (3.10) can be calculated as follows:

$$\sum_{n=1}^N \mathcal{J}_n \cdot \mathbf{e}_n = \mathcal{J}_\epsilon \cdot \epsilon \quad (3.14)$$

Note that the dimension of the product is  $D_w \times 1$ .

Using the above notation the **Gauss-Newton** equation can be written in the following simple form:

$$\Delta \mathbf{w} = -(\mathcal{J}_\epsilon \cdot \mathcal{J}_\epsilon^T)^{-1} \cdot \mathcal{J}_\epsilon \cdot \epsilon \quad (3.15)$$

Note that the dimensionality of the  $\Delta \mathbf{w}$  and  $\mathbf{w}$  vectors is  $D_w$ .

The Hessian matrix of second derivatives and its Jacobian approximation can become a non-invertible (singular, non-full rank) matrix for some values of parameters. Recall that for the single-variable function its second derivative is zero in the point of inflection, at which the curve changes from being concave to convex, or vice versa.

In order to prevent the singularity of the Hessian matrix at inflection points/lines stopping the iterations, in the **Levenberg-Marquardt algorithm** a small constant  $\mu$  is added to the diagonal of the matrix to be inverted. This results in the following expression for the increment of the parameter vector:

$$\Delta \mathbf{w} = -(\mathcal{J}_\epsilon \cdot \mathcal{J}_\epsilon^T + \mu I)^{-1} \cdot \mathcal{J}_\epsilon \cdot \epsilon \quad (3.16)$$

## 4 More on non-linear learning algorithms

This section is a conceptual continuation of sec. 3. Neural networks are the most popular and well-studied example of a non-linear mapping and related learning or optimization procedures.

### 4.1 Why gradient-decent algorithms are slow

- The basic pattern-based back-propagation learning law is a gradient-descent algorithm based on the estimation of the gradient of the instantaneous sum-squared error for each layer:

$$\Delta W(n) = -\eta \cdot \nabla_W E(n) = \eta \cdot \boldsymbol{\delta}(n) \cdot \mathbf{x}^T(n) \quad (4.1)$$

Such an algorithm is slow for a few reasons:

- It uses an instantaneous sum-squared error  $E(W, n)$  to minimise the mean squared error,  $L(W)$ , over the training epoch.
- The gradient of the instantaneous sum-squared error is not a good estimate of the gradient of the mean squared error.
- Therefore, satisfactory minimisation of this error typically requires many repetitions of the training epochs.
- It is a first-order minimisation algorithm which is based on the first-order derivatives (a gradient). Faster algorithms utilise also the second derivatives (the Hessian matrix) as discussed in sec. 3.
- The error back propagation, which is conceptually very interesting, serialises computations on the layer by layer basis.

A general problem is that the mean squared error,  $L(W)$ , is a relatively complex surface in the weight space, possibly with many local minima, flat sections, narrow irregular valleys, and saddle points, therefore, it is difficult to navigate directly to its minimum.

#### 4.1.1 Examples of error surfaces

Consider an example of a function of two weights,  $L(w_1, w_2)$  representing a possible mean-squared error or a Loss function. In Figure 4–1 the surface plot and the contour map are shown.

In the plots note that the loss function has the local and global minima, and a saddle point. Depending on initialisation, during learning we might end up either in a local or global minimum. Complexity of the error surface is the main reason that behaviour of a simple gradient descent minimisation algorithm can be very complex often with oscillations around a local minimum.

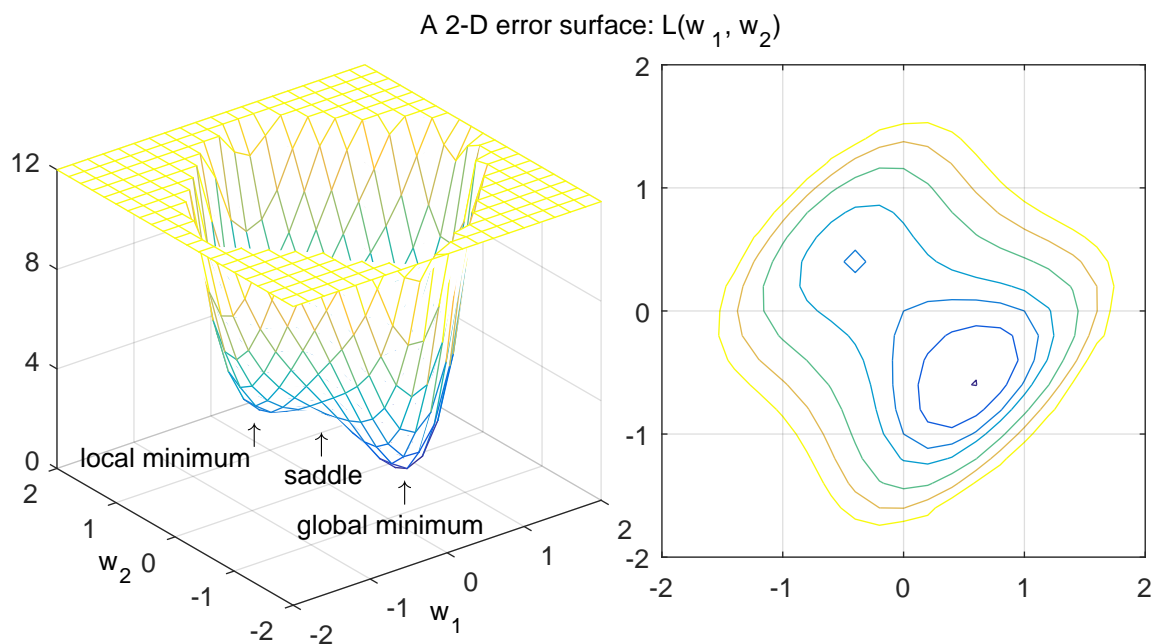
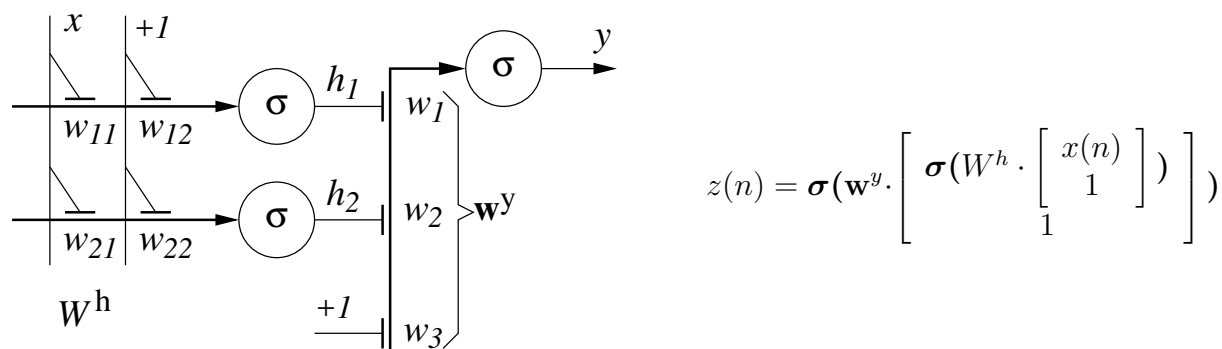


Figure 4-1: An example of a Loss function of two weights

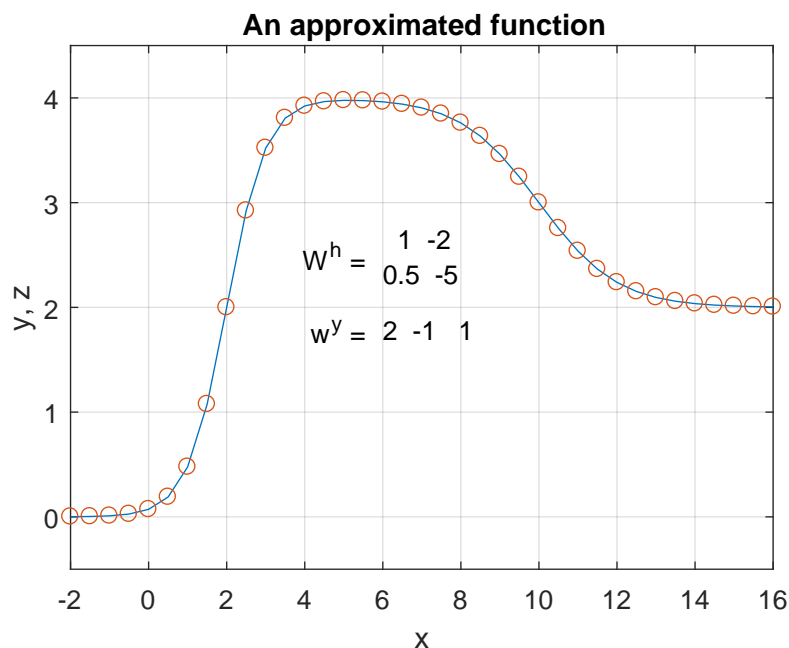
In order to gain more insight into the shape of the error surfaces let us consider a simple two-layer network approximating a single-variable function similar to that considered in sec. 6.8.



If, for example, the weights are as follows:

$$W^h = \begin{bmatrix} 1 & -2 \\ 0.5 & -5 \end{bmatrix} \quad \text{and} \quad \mathbf{w}^y = \begin{bmatrix} 2 & -1 & 1 \end{bmatrix}$$

then, the network approximate the following function:



In order to obtain the error surface, we will vary parameters  $\mathbf{w} = [W^h \ \mathbf{w}^y]$  and calculate  $L(\mathbf{w})$  for the selected inputs  $X$ . The error function  $L(\mathbf{w})$  depends on 4+3 parameters and is an 8-dimensional object, hence difficult to visualise. Therefore we will vary only a pair of selected weights at a time. The resulting surfaces of the loss function are shown in Figure 4–2.

Note that the surfaces are very far away from an ideal second order paraboloidal shapes. Note also that finding the minimum of such complex function is very sensitive to the initial position, learning rate and the direction of movement.

#### 4.1.2 Illustration of sensitivity to a learning rate

(Figures 9.1, 9.2, 9.3, 12.6, 12.7, 12.8 from: M.T. Hagan, H. Demuth, M. Beale, *Neural Network Design*, PWS Publishing, 1996)

For a linear network, aka Adaline, when the error surface is paraboloidal, the maximum stable learning rate can be shown to be inversely proportional to the largest eigenvalue of the input correlation matrix,  $R$ .

As an illustration we consider the case when  $\eta_{max} = 0.04$  and observe the learning trajectory on the error surface for a linear case as in Figure 4–3.

In the case of the nonlinear neural network, with the loss surfaces as illustrated in Figure 4–2, learning trajectories are even more complex. In Figure 4–4 there are examples of possible learning trajectories in for the gradient descent backpropagation algorithm in the batch mode.

## 4.2 Heuristic Improvements to the Back-Propagation Algorithm

As discussed sec. 3, the significant improvement to the learning speed can be achieved by replacing the gradient decent algorithm by the second order algorithms using second derivatives



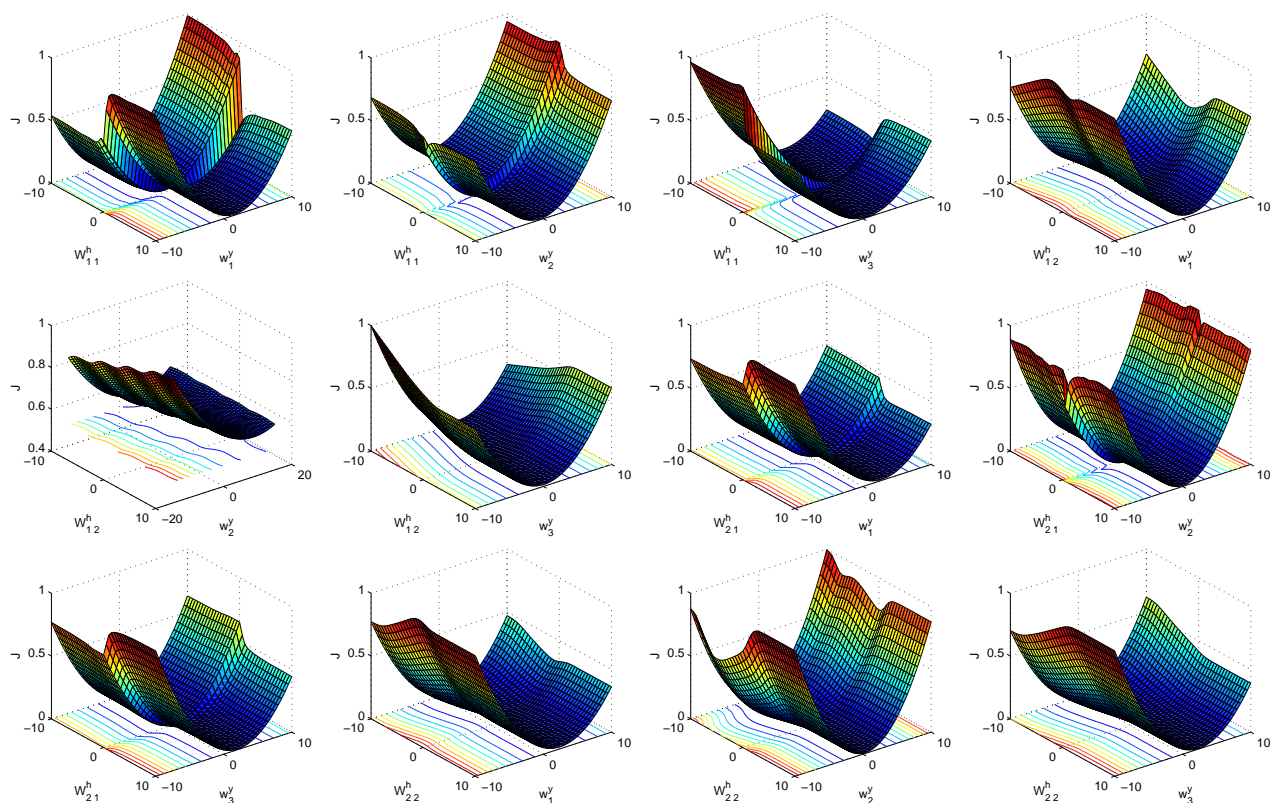


Figure 4-2: Projections of a loss function of seven parameters on the 2-dimensional space.

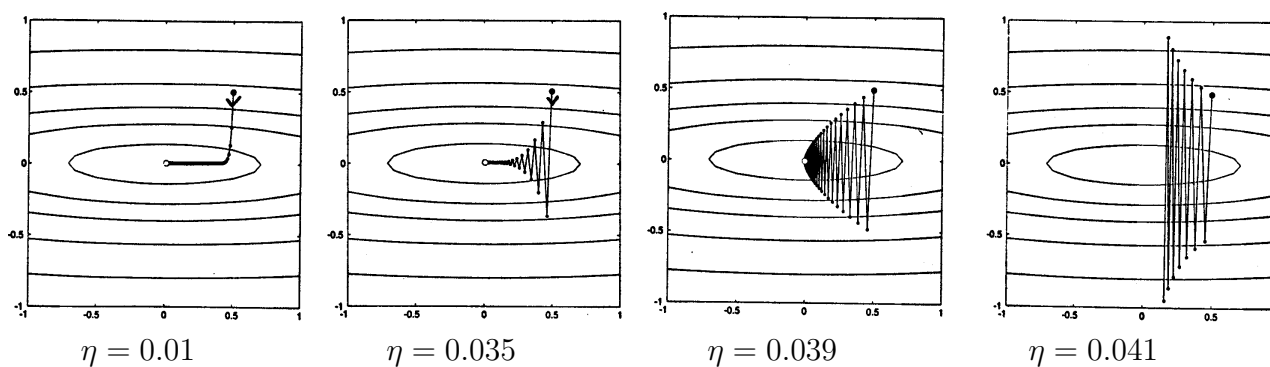
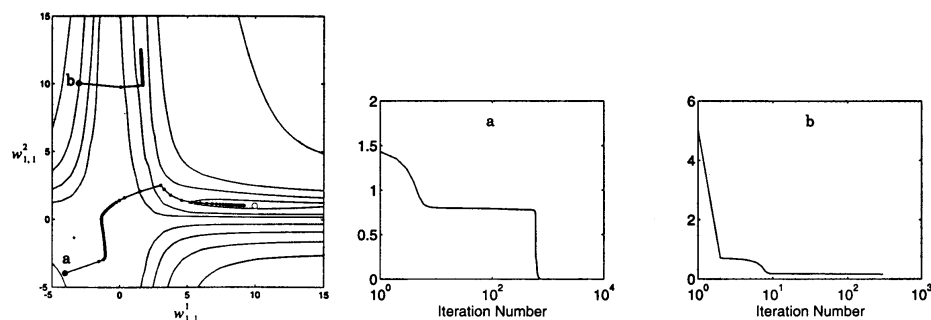


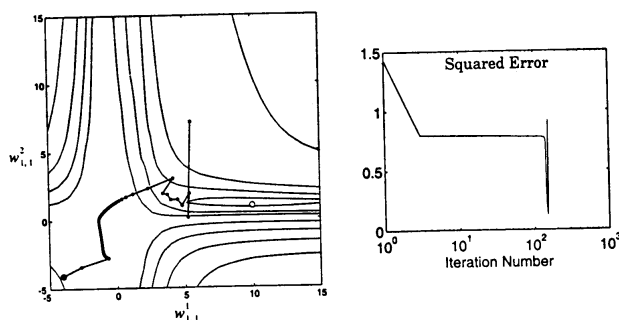
Figure 4-3: Example of learning trajectories for various learning rate for a linear network

of the loss function.

As the calculation of the Hessian matrix of second derivatives is typically computationally demanding, in this section, we will consider heuristic group of methods aiming at improvement of the learning speed of the basic gradient descent algorithm without using the second derivatives. These methods do not directly address the inherent weaknesses of the back-propagation algorithm, but aim at improvement of the behaviour of the algorithm by making modifications to its form or parameters.



Plots on the right shows the error versus the iteration number.



Trajectory for the learning rate too large

Figure 4–4: Example of learning trajectories for the gradient descent backpropagation algorithm in the batch mode

#### 4.2.1 The momentum term

One of the simple method to avoid an error trajectory in the weight space being oscillatory is to add to the weight update a momentum term. Such a term is proportional to the weight update at the previous step.

$$\Delta W(n) = \eta \cdot \delta(n) \cdot \mathbf{x}^T(n) + \alpha \cdot \Delta W(n-1), \quad 0 < \alpha < 1 \quad (4.2)$$

where  $\alpha$  is a momentum term parameter.

Such modification to the steepest descend learning law acts as a low-pass filter smoothing the error trajectory as shown in Figure 4–5. As a result it is possible to apply higher learning rate,  $\eta$ .

#### 4.2.2 Adaptive learning rate

One of the ways of increasing the convergence speed, that is, to move faster downhill to the minimum of the mean-squared error,  $J(W)$ , is to vary adaptively the learning rate parameter,  $\eta$ . A related example is shown in Figure 4–6.

A typical strategy is based on monitoring the rate of change of the mean-squared error and can be described as follows.

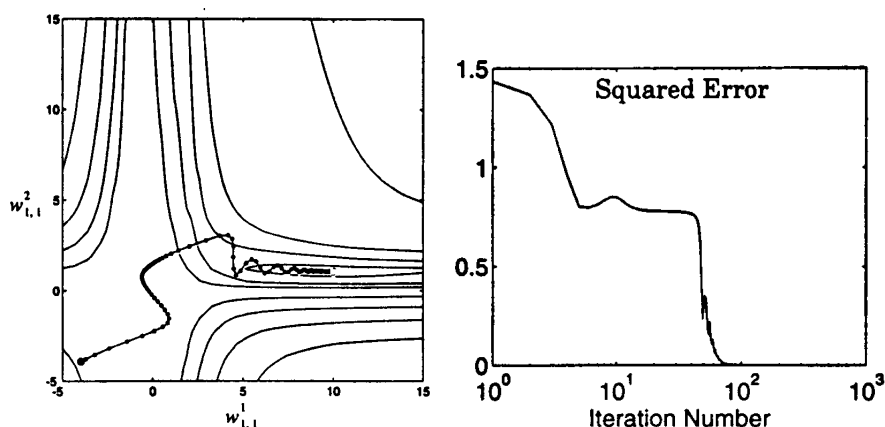


Figure 4-5: Example of learning trajectories with momentum term

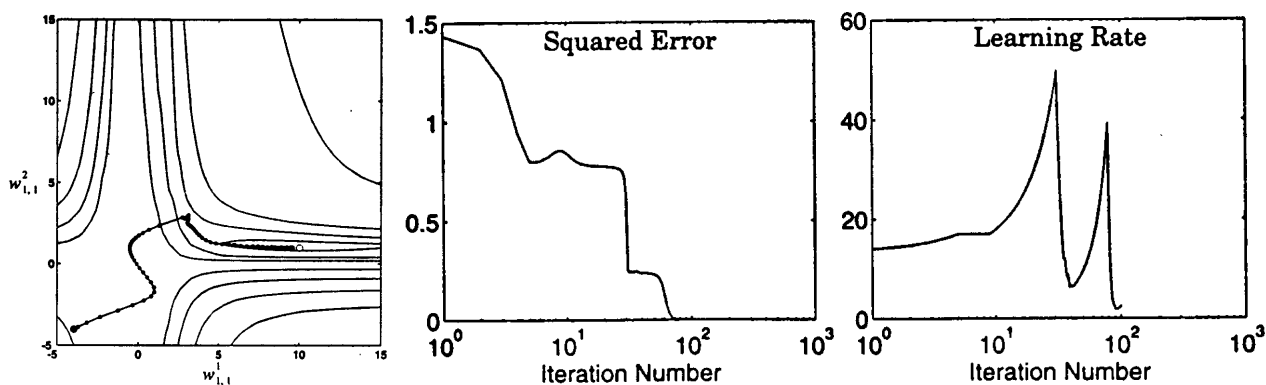


Figure 4-6: Example of learning trajectories with adaptive learning rate

If  $L$  is decreasing consistently, that is,  $\nabla L$  is negative for a prescribed number of steps, then the learning rate is increased linearly:

$$\eta(n+1) = \eta(n) + a, \quad a > 0 \quad (4.3)$$

If the error has increased, ( $\nabla L > 0$ ), the learning rate is exponentially reduced.

$$\eta(n+1) = b \cdot \eta(n), \quad 0 < b < 1 \quad (4.4)$$

The rationale behind the exponential reduction of the value of the learning rate is that, in general, increasing the value of the learning rate the learning tends to become unstable which is indicated by an increase in the value of the error function. Therefore it is important to quickly reduce  $\eta$ .

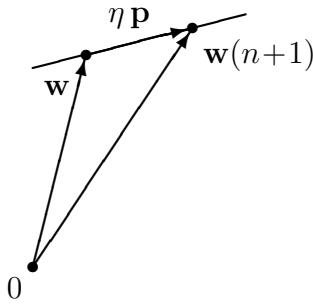
### 4.3 Line search minimisation procedures

Another approach is to try to find an optimal learning rate  $\eta$  that minimizes the loss function wrt to  $\eta$ . This is equivalent to the line search optimization procedure.

Gradient descent minimization procedures are based on updating the weight vector

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta \mathbf{p}(n) \quad (4.5)$$

where the vector  $\mathbf{p}(n)$  describes the direction of modification of the weight vector. The vector  $\mathbf{p}$  is typically equal to the negative gradient of the error function



$$\mathbf{p}(n) = -\mathbf{g}(n), \quad \text{where } \mathbf{g}(n) = \nabla L(\mathbf{w}(n))$$

Note that the next value of weight vector,  $\mathbf{w}(n+1)$ , is obtained from the current value of weight vector,  $\mathbf{w}$ , by moving it along the direction of a vector,  $\mathbf{p}$ . We can now find an optimal value of  $\eta$  for which the loss function

$$L(\mathbf{w}(n+1)) = L(\mathbf{w}(n) + \eta \mathbf{p}) \quad (4.6)$$

is minimised. The optimal  $\eta$  is typically found through a search procedure along the direction  $\mathbf{p}$  as illustrated above.

The next step is to calculate the partial derivative of  $L$  with respect to  $\eta$ :

$$\frac{\partial L(\mathbf{w}(n+1))}{\partial \eta} = \frac{\partial L(\mathbf{w}(n+1))}{\partial \mathbf{w}(n+1)} \frac{\partial \mathbf{w}(n+1)}{\partial \eta} = \frac{\partial L(\mathbf{w}(n+1))}{\partial \mathbf{w}(n+1)} \eta \mathbf{p}^T = \eta \nabla L(\mathbf{w}(n+1)) \cdot \mathbf{p}^T \quad (4.7)$$

For the optimal value of  $\eta$ , this derivative needs to be zero and we have the following relationship

$$\nabla L(\mathbf{w}(n+1)) \cdot \mathbf{p}^T = 0 \quad (4.8)$$

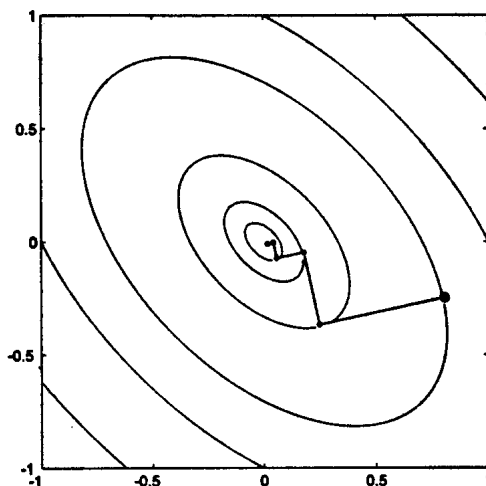
and consequently

$$\mathbf{g}(n+1) \cdot \mathbf{p}^T = 0 \quad (4.9)$$

It states, unexpectedly, that for the optimal value of  $\eta$ , the next estimate of the gradient,  $\mathbf{g}(n+1) = \nabla L(\mathbf{w}(n+1))$ , is to be orthogonal to the current search direction,  $\mathbf{p}$ .

This means, in particular, that if we combine the **line minimisation** technique with the **gradient descent** algorithm when we move in the direction opposite to the gradient,  $\mathbf{p} = -\mathbf{g}$ ,

then we will be descending along the zig-zag line, each segment being orthogonal to the next one as illustrated in the following Figure.



In order to smooth the descend direction, the basic line minimization method is replaced with the conjugate gradient algorithms.

#### 4.4 Conjugate Gradient Algorithms

The conjugate gradient algorithms also involved the line optimisation with respect to  $\eta$ , but in order to avoid the zig-zag movement through the error surface, the next search direction,  $\mathbf{p}(n+1)$ , instead of being exactly orthogonal to the gradient, tries to maintain the current search direction,  $\mathbf{p}(n)$ , namely

$$\mathbf{p}(n+1) = -\mathbf{g}(n) + \beta(n)\mathbf{p}(n) \quad (4.10)$$

where scalar  $\beta(n)$  is selected in such a way that the directions  $\mathbf{p}(n+1)$  and  $\mathbf{p}(n)$  are **conjugate with respect to the Hessian matrix**,  $\nabla^2 L(\mathbf{w}) = H$  (the matrix of all second derivatives of  $J$ ), that is,

$$\mathbf{p}(n+1) \cdot H \cdot \mathbf{p}^T(n) = 0 \quad (4.11)$$

In practice, the Hessian matrix is not being calculated and the following three approximate choices of  $\beta(n)$  are the most commonly used

- Hestenes-Steifel formula

$$\beta(n) = \frac{(\mathbf{g}(n) - \mathbf{g}(n-1)) \cdot \mathbf{g}^T(n)}{(\mathbf{g}(n) - \mathbf{g}(n-1)) \cdot \mathbf{p}^T(n-1)} \quad (4.12)$$

- Fletcher-Reeves formula

$$\beta(n) = \frac{\mathbf{g}(n) \cdot \mathbf{g}^T(n)}{\mathbf{g}(n-1) \cdot \mathbf{g}^T(n-1)} \quad (4.13)$$

- Polak-Ribière formula

$$\beta(n) = \frac{(\mathbf{g}(n) - \mathbf{g}(n-1)) \cdot \mathbf{g}^T(n)}{\mathbf{g}(n-1) \cdot \mathbf{g}^T(n-1)} \quad (4.14)$$

In summary, the conjugate gradient algorithm involves:

- initial search direction,  $\mathbf{p}(0) = -\mathbf{g}(0)$ ,
- line minimisation with respect of  $\eta$ ,
- calculation of the next search direction as in eqn (4.10),
- $\beta$  from one of the above formulae.

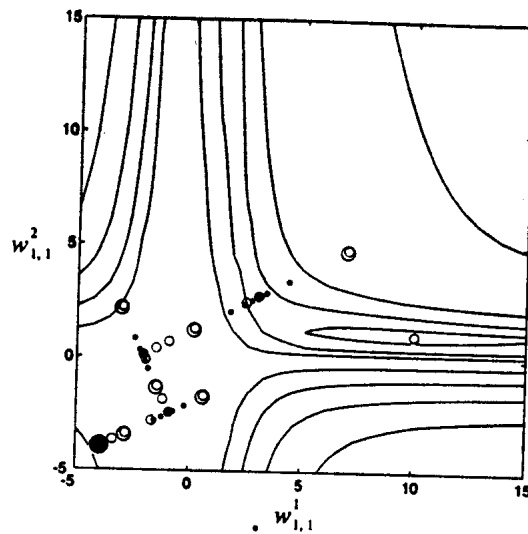


Figure 12.15 Intermediate Steps of CGBP

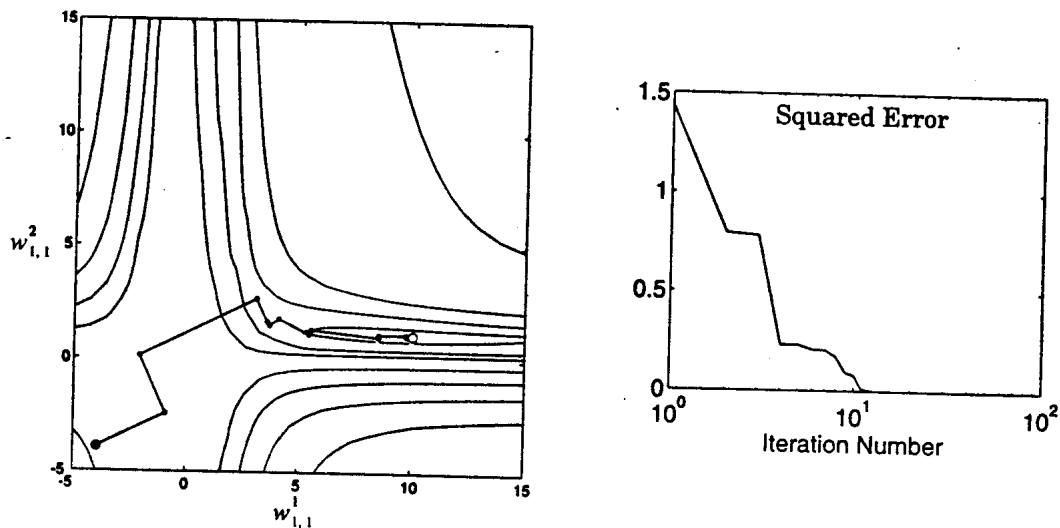


Figure 12.16 Conjugate Gradient Trajectory

## 4.5 The Adam learning/optimization algorithm

Refer to the original paper: Diederik P. Kingma and Jimmy Lei Ba, Adam: A Method for Stochastic Optimization, ICLR 2015 ( arXiv:1412.6980v9 [cs.LG] 30 Jan 2017 )

quote from the above:

---

**Algorithm 1:** *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation.  $g_t^2$  indicates the elementwise square  $g_t \odot g_t$ . Good default settings for the tested machine learning problems are  $\alpha = 0.001$ ,  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$  and  $\epsilon = 10^{-8}$ . All operations on vectors are element-wise. With  $\beta_1^t$  and  $\beta_2^t$  we denote  $\beta_1$  and  $\beta_2$  to the power  $t$ .

---

**Require:**  $\alpha$ : Stepsize

**Require:**  $\beta_1, \beta_2 \in [0, 1)$ : Exponential decay rates for the moment estimates

**Require:**  $f(\theta)$ : Stochastic objective function with parameters  $\theta$

**Require:**  $\theta_0$ : Initial parameter vector

$m_0$  0 (Initialize 1<sup>st</sup> moment vector)

$v_0$  0 (Initialize 2<sup>nd</sup> moment vector)

$t$  0 (Initialize timestep)

**while**  $\theta_t$  not converged **do**

$t$   $t + 1$

$g_t$   $\nabla_{\theta} f_t(\theta_{t-1})$  (Get gradients w.r.t. stochastic objective at timestep  $t$ )

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$  (Update biased first moment estimate)

$v_t$   $\beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$  (Update biased second raw moment estimate)

$\hat{m}_t$   $m_t / (1 - \beta_1^t)$  (Compute bias-corrected first moment estimate)

$\hat{v}_t$   $v_t / (1 - \beta_2^t)$  (Compute bias-corrected second raw moment estimate)

$\theta_t$   $\theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$  (Update parameters)

**end while**

**return**  $\theta_t$  (Resulting parameters)

---

## 5 Expectation Maximization Algorithm in Point Set Registration

In previous sections we have performed optimization by reducing errors. In this section, we present an alternative approach, often referred to as a Bayesian approach, in which we maximize expectations rather than minimize errors.

In particular, in this section we study some popular concepts from the non-linear mapping area. In particular, we investigate

- Gaussian Mixture Models (GMMs)
- Expectation Maximization (EM) Algorithm
- Affine and Rigid Point Set Registration
- Coherent Point Drift Algorithm

We will generalize the generic mapping problem formulation as described in sec. 1.2, to the case when the number of points in the sets  $X$  and  $Y$  are different. Therefore, we cannot use the errors as in eqn (1.4), instead of we need the probabilistic re-formulation of the mapping problem. We consider details of point set registration following formulation closely similar to that presented in the paper: Andriy Myronenko and Xubo Song, *Point Set Registration: Coherent Point Drift*. IEEE Tran. PAMI, 32(12), 2010.

### 5.1 Point Set Registration Fundamentals

We have to map (or register)  $N$  points  $X$  to  $N$  points  $Z$  that are as close as possible to  $M$  points  $Y$ . Unlike in sec. 1.2, all points have dimensionality  $D$ . Modifying notation from sec. 1.2 we define:

$$X = [ \mathbf{x}_1 \cdots \mathbf{x}_n \cdots \mathbf{x}_N ] \xrightarrow{F} Z = [ \mathbf{z}_1 \cdots \mathbf{z}_n \cdots \mathbf{z}_N ] \approx Y = [ \mathbf{y}_1 \cdots \mathbf{y}_m \cdots \mathbf{y}_M ] \quad (5.1)$$

and the parameterize mapping:

$$\mathbf{z}_n = F(\mathbf{x}_n; \mathbf{w}) \text{ or } Z = F(X; \mathbf{w}) \text{ such that } Z \approx Y \quad (5.2)$$

that is, the original point set,  $X$ , is mapped into the aligned point set,  $Z$ , that approximates the target point set  $Y$ . We illustrate the arrangement for  $D = 2$  in Fig. 5-1

In addition to finding parameters  $\mathbf{w}$  of the mapping  $F$ , we have to find the correspondence between points  $X$  and  $Y$  since their numbers are different. To solve the correspondence problem, we create a Gaussian Mixture Model (GMM) that generates points  $\mathbf{y}_m$  given points  $\mathbf{z}_n$  as the centroids of the GMM. The GMM probability density function is defined as:

$$p(\mathbf{y}_m) = \sum_{n=1}^{N+1} P(n)p(\mathbf{y}_m|n) \quad (5.3)$$



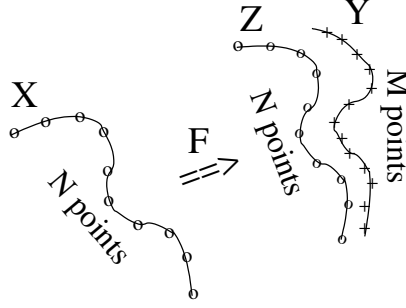


Figure 5–1: Illustration of mapping/registration of 2D points  $X$  and  $Y$  using function  $F$  as in eqn (5.2)

**for**  $n = 1, \dots, N$

we use Gaussian distributions centered around  $\mathbf{z}_n$  and with equal isotropic covariances  $\sigma^2$

$$p(\mathbf{y}_m|n) = \frac{1}{(2\pi\sigma^2)^{D/2}} \exp\left(-\frac{\|\mathbf{y}_m - \mathbf{z}_n\|^2}{2\sigma^2}\right) = \frac{1}{(2\pi\sigma^2)^{D/2}} \exp\left(-\frac{h_{nm}}{2\sigma^2}\right) \quad (5.4)$$

where  $h_{nm}$  are the squares of Euclidian distances between points  $\mathbf{y}_m$  and  $\mathbf{z}_n$ :

$$h_{nm} = \|\mathbf{y}_m - \mathbf{z}_n\|^2 = (\mathbf{z}_n - \mathbf{y}_m)^T \cdot (\mathbf{z}_n - \mathbf{y}_m), \quad H = \{h_{nm}\}_{N \times M} \quad (5.5)$$

and  $H$  is an  $N \times M$  matrix collecting all the squares of distances between points  $\mathbf{z}_n$  and  $\mathbf{y}_m$ . Note that the all the differences  $\mathbf{z}_n - \mathbf{y}_m$  create a tensor (3-dimensional array) of the size  $N \times M \times D$ .

We also assume equal membership probabilities

$$P(n) = \frac{1}{N} \quad (5.6)$$

**for**  $n = N + 1$

we have an additional uniform distribution to account for noise and outliers

$$p(\mathbf{y}_m|N + 1) = \frac{1}{M} \quad (5.7)$$

For each  $\mathbf{y}_m$ , the uniform distribution contributes to the model with the weight  $0 \leq \omega \leq 1$  so that the complete mixture model of eqn (5.3) takes the following form:

$$p(\mathbf{y}_m) = \frac{\omega}{M} + \frac{1 - \omega}{N} \sum_{n=1}^N p(\mathbf{y}_m|n) \quad (5.8)$$

where  $p(\mathbf{y}_m|n)$  are Gaussians as in eqn (5.4).

Now, the first task is to find the optimal parameters  $\mathbf{w}$  of the mapping  $Z = F(X; \mathbf{w})$  and  $\sigma^2$  of the distribution  $p(\mathbf{y}_m|n)$ . We can do that by minimising the negative log-likelihood function:

$$\mathcal{E}(\mathbf{w}, \sigma^2) = - \sum_{m=1}^M \log p(\mathbf{y}_m) = - \sum_{m=1}^M \log \sum_{n=1}^{N+1} P(n)p(\mathbf{y}_m|n) \quad (5.9)$$

The second task is to establish the correspondence between points  $\mathbf{z}_n$  ( $\mathbf{x}_n$ ) and  $\mathbf{y}_m$ . We define it as the posterior probability of the GMM centroids  $\mathbf{z}_n$  given the points  $\mathbf{y}_m$ , that is, as :

$$P(n|\mathbf{y}_m) = \frac{P(n)p(\mathbf{y}_m|n)}{p(\mathbf{y}_m)} \quad (5.10)$$

## 5.2 Expectation Maximization (EM) Algorithm

Now, we use the **Expectation Maximization** (EM) algorithm in which, the problem of minimisation of the function  $\mathcal{E}(\mathbf{w}, \sigma^2)$  of eqn (5.9) is replaced by minimisation of the expectation of the complete log-likelihood function, namely

$$Q = - \sum_{m=1}^M \sum_{n=1}^{N+1} P(n|\mathbf{y}_m) \log(P(n)p(\mathbf{y}_m|n)) \quad (5.11)$$

Expanding the inner sum we get:

$$\begin{aligned} & - \sum_{n=1}^N P(n|\mathbf{y}_m) \log(P(n)p(\mathbf{y}_m|n)) - P(N+1|\mathbf{y}_m) \log(P(N+1)p(\mathbf{y}_m|N+1)) \\ &= - \sum_{n=1}^N P(n|\mathbf{y}_m) \log \left( \frac{1-\omega}{N} \frac{1}{(2\pi\sigma^2)^{D/2}} \exp\left(-\frac{h_{nm}}{2\sigma^2}\right) \right) - P(N+1|\mathbf{y}_m) \log \left( \frac{\omega}{M} \right) \\ &= \sum_{n=1}^N P(n|\mathbf{y}_m) \left( \log \left( \frac{N}{1-\omega} (2\pi\sigma^2)^{\frac{D}{2}} \right) + \frac{h_{nm}}{2\sigma^2} \right) + P(N+1|\mathbf{y}_m) \log \left( \frac{\omega}{M} \right) \\ &= \frac{1}{2\sigma^2} \sum_{n=1}^N P(n|\mathbf{y}_m) \cdot h_{nm} + \sum_{n=1}^N P(n|\mathbf{y}_m) \left( \log \frac{N(2\pi)^{\frac{D}{2}}}{1-\omega} + \frac{D}{2} \log \sigma^2 \right) + P(N+1|\mathbf{y}_m) \log \left( \frac{\omega}{M} \right) \end{aligned}$$

Before we put this result back into the final form of the objective function  $Q$ , we note that we can ignore terms independent of  $\mathbf{w}$ , hidden in  $\mathbf{d}_{nm}$ , and  $\sigma^2$ . Also, we can collect the posterior probabilities into an  $N \times M$  matrix  $P$  with entries:

$$p_{nm} = P(n|\mathbf{y}_m), \quad P = \{p_{nm}\}_{N \times M} \quad (5.12)$$

With that the objective function of eqn (5.11) can be written in the form

$$Q(\mathbf{w}, \sigma^2) = \frac{1}{2\sigma^2} \sum_{m=1}^M \sum_{n=1}^N p_{nm} h_{nm} + \frac{\bar{p}D}{2} \log \sigma^2 \quad (5.13)$$

where  $\bar{p}$  is the sum of all posterior probabilities

$$\bar{p} = \sum_{m=1}^M \sum_{n=1}^N P(n|\mathbf{y}_m) = \mathbf{1}_N^T \cdot P \cdot \mathbf{1}_M = \mathbf{1}_M^T \cdot P^T \cdot \mathbf{1}_N \quad (5.14)$$

Note that  $\bar{p} \leq M$  and  $\bar{p} = M$  only if  $\omega = 0$ . We denote by  $\mathbf{1}_M$  a column vector of a  $M$  ones. Such a vector of ones is convenient to describe summation and also replication of another vector.

The posterior probabilities can be evaluated using eqn (5.10) as:

$$p_{nm} = P(n|\mathbf{y}_m) = \frac{\frac{1-\omega}{N} \frac{1}{(2\pi\sigma^2)^{D/2}} \exp(-\frac{h_{nm}}{2\sigma^2})}{\frac{\omega}{M} + \frac{1-\omega}{N} \frac{1}{(2\pi\sigma^2)^{D/2}} \sum_{k=1}^N \exp(-\frac{h_{km}}{2\sigma^2})}$$

and can be re-written as:

$$p_{nm} = P(n|\mathbf{y}_m) = \frac{\exp(-\frac{h_{nm}}{2\sigma^2})}{c + \sum_{k=1}^N \exp(-\frac{h_{km}}{2\sigma^2})} \quad (5.15)$$

where

$$c = \frac{\omega}{1-\omega} \frac{N}{M} (2\pi\sigma^2)^{\frac{D}{2}} \quad (5.16)$$

or, in a matrix form

$$P = E /_r (c + \mathbf{1}_N \cdot E) \quad , \quad E = \exp(-\frac{H}{2\sigma^2}) \quad (5.17)$$

where the matrix  $H$  is specified in eqn (5.5) and  $/_r$  denotes division of a matrix by a row vector on a row-by-row basis.

We are ready to consider minimization of the objective function (5.13) using the EM algorithm. The algorithm consists of two steps:

**E-step:** In this step we calculate the posterior probabilities  $p_{nm}$  for the current values of parameters  $\mathbf{w}$  and  $\sigma^2$ ,

**M-step:** In this step we assume that the posterior probabilities  $p_{nm}$  are constant and minimise the objective function wrt to “new” values of parameters  $\mathbf{w}$  and  $\sigma^2$ . Having  $p_{nm}$  constant, makes calculation of the gradient of the objective function  $Q(\mathbf{w}, \sigma^2)$  simpler.

The E and M steps are repeated until the convergence is attained.

We can start with calculation of the M-step-optimal value of the variance  $\sigma^2$  that minimizes the objective function (5.13). Assuming as explained in the E-step that the posterior probabilities are calculated for the old value of  $\sigma^2$ , the objective function is reduced to the form:

$$Q = \frac{\alpha}{2\sigma^2} + \frac{\bar{p}D}{2} \log \sigma^2 \quad (5.18)$$

where

$$\alpha = \sum_{m=1}^M \sum_{n=1}^N p_{nm} h_{nm} = \text{tr}(P \cdot H^T) = \text{tr}(H \cdot P^T) \quad (5.19)$$

where  $\text{tr}(M)$  is a sum of the diagonal elements of a square matrix  $M$ . You can refresh your knowledge of the trace from:

[https://en.wikipedia.org/wiki/Trace\\_\(linear\\_algebra\)](https://en.wikipedia.org/wiki/Trace_(linear_algebra)).

Calculating the derivative  $\frac{\partial Q}{\partial \sigma^2}$  and equating it to zero, gives the optimal value of  $\sigma^2$

$$\sigma^2 = \frac{\alpha}{\bar{p}D} \quad (5.20)$$

Next we can calculate the optimal value of the parameters  $\mathbf{w}$ . Without making any assumption yet about the form of the mapping  $\mathbf{z}_n = F(\mathbf{x}_n)$ , we can express the gradient of the objective function wrt parameters  $\mathbf{w}$  as (Note that you have scalar, vectors and matrices which enforces specific order of multiplications. It might be useful to go back to sec. 2 )

$$\frac{\partial Q}{\partial \mathbf{w}} = \frac{1}{2\sigma^2} \sum_{m=1}^M \sum_{n=1}^N p_{nm} \frac{\partial \mathbf{z}_n^T}{\partial \mathbf{w}} \frac{\partial h_{nm}}{\partial \mathbf{z}_n}$$

Noting that  $h_{nm} = (\mathbf{z}_n - \mathbf{y}_m)^T \cdot (\mathbf{z}_n - \mathbf{y}_m)$  we can write:

$$\frac{\partial Q}{\partial \mathbf{w}} = \frac{1}{\sigma^2} \sum_{m=1}^M \sum_{n=1}^N \left( p_{nm} \frac{\partial \mathbf{z}_n^T}{\partial \mathbf{w}} (\mathbf{z}_n - \mathbf{y}_m) \right) \quad (5.21)$$

### 5.3 Affine point set registration

Firstly, we consider the simplest case of the affine transformation, used to register points  $X$  and  $Y$ . As in sec. 2, we have

$$\mathbf{z}_n = A \cdot \mathbf{x}_n + \mathbf{b} \quad \text{or} \quad Z = A \cdot X + \mathbf{b} \cdot \mathbf{1}_N^T \quad (5.22)$$

It is now convenient to do optimization wrt the vector  $\mathbf{b}$ . Hence, adopting eqn (5.21) the gradient can be written as:

$$\frac{\partial Q}{\partial \mathbf{b}} = \frac{1}{\sigma^2} \sum_{m=1}^M \sum_{n=1}^N p_{nm} (\mathbf{z}_n - \mathbf{y}_m)$$

since

$$\frac{\partial \mathbf{z}_n^T}{\partial \mathbf{b}} = I_D$$

where  $I_D$  is the  $D \times D$  identity matrix. Equating the gradient to zero gives:

$$\sum_{m=1}^M \sum_{n=1}^N \mathbf{z}_n p_{nm} = \sum_{n=1}^N \sum_{m=1}^M \mathbf{y}_m p_{nm}$$

re-writing in a matrix form gives:

$$Z \cdot P \cdot \mathbf{1}_M = Y \cdot P^T \cdot \mathbf{1}_N$$

or

$$Z \cdot \mathbf{p}_c = Y \cdot \mathbf{p}_r$$

where

$$\mathbf{p}_c = P \cdot \mathbf{1}_M \quad \text{and} \quad \mathbf{p}_r = P^T \cdot \mathbf{1}_N \quad (5.23)$$

are  $N \times 1$  and  $M \times 1$  vectors being sums of columns and rows of the matrix  $P$ , respectively. Using eqn (5.22) gives the expression for the optimal value of the vector  $\mathbf{b}$  in the current M-step of the EM algorithm:

$$\mathbf{b} \cdot \mathbf{1}_N^T \cdot P \cdot \mathbf{1}_M = Y \cdot \mathbf{p}_r - A \cdot X \cdot \mathbf{p}_c$$

or

$$\mathbf{b} \cdot \bar{\mathbf{p}} = Y \cdot \mathbf{p}_r - A \cdot X \cdot \mathbf{p}_c$$

and finally

$$\mathbf{b} = \mathbf{y}_r - A \cdot X \cdot \hat{\mathbf{p}}_c \quad (5.24)$$

where  $\bar{\mathbf{p}}$  is defined in eqn (5.14) and

$$\hat{\mathbf{p}}_c = \mathbf{p}_c / \bar{p} \quad , \quad \hat{\mathbf{p}}_r = \mathbf{p}_r / \bar{p} \quad \text{and} \quad \mathbf{y}_r = Y \cdot \hat{\mathbf{p}}_r \quad (5.25)$$

Note that the  $D \times 1$  vector  $\mathbf{y}_c$  is a weighted sum of all vectors  $\mathbf{y}_m$ . Now, we can eliminate the vector  $\mathbf{b}$  from the mapping (5.22) to get:

$$Z = A \cdot X + \mathbf{y}_r \cdot \mathbf{1}_N^T - A \cdot X \cdot \hat{\mathbf{p}}_c \cdot \mathbf{1}_N^T$$

Re-arranging the terms, we get

$$Z = A \cdot X \cdot (I_N - \hat{\mathbf{p}}_c \cdot \mathbf{1}_N^T) + \mathbf{y}_r \cdot \mathbf{1}_N^T$$

and finally

$$Z = A \cdot \hat{X} + \mathbf{y}_r \cdot \mathbf{1}_N^T \quad \text{or} \quad \mathbf{z}_n = A \cdot \hat{\mathbf{x}}_n + \mathbf{y}_r \quad (5.26)$$

where

$$\hat{X} = \{\hat{\mathbf{x}}_n\}_{D \times N} = X \cdot (I_N - \hat{\mathbf{p}}_c \cdot \mathbf{1}_N^T) \quad (5.27)$$

Eqn (5.26) is modification of the original mapping given in eqn (5.22) after substitution of the M-step optimal value of  $\mathbf{b}$ . Now, we can continue calculation of the M-step optimal value of the matrix  $A$ , starting with the gradient of eqn (5.21) knowing that

$$\mathbf{w} = \text{vec}(A) = \downarrow A \quad (5.28)$$

Now, we can follow considerations of sec. 2.2 and, in particular, eqns (2.12) – (2.18) and expand the eqn (5.21) in the following way:

$$\frac{\partial \mathbf{z}_n^T}{\partial \mathbf{w}} (\mathbf{z}_n - \mathbf{y}_m) = (\hat{\mathbf{x}}_n \otimes I_D) \cdot (\mathbf{z}_n - \mathbf{y}_m) = \hat{\mathbf{x}}_n \otimes (\mathbf{z}_n - \mathbf{y}_m) = \downarrow ((\mathbf{z}_n - \mathbf{y}_m) \cdot \hat{\mathbf{x}}_n^T)$$

Equating the gradient to zero results in the following set of linear equations for the matrix  $A$ :

$$\sum_{m=1}^M \sum_{n=1}^N p_{nm} \downarrow ((A \cdot \hat{\mathbf{x}}_n + \mathbf{y}_r - \mathbf{y}_m) \cdot \hat{\mathbf{x}}_n^T) = 0$$

where  $\hat{\mathbf{x}}_n$  are defined in eqn (5.27). Further we have

$$\sum_{m=1}^M \sum_{n=1}^N p_{nm} \downarrow (A \cdot \hat{\mathbf{x}}_n \cdot \hat{\mathbf{x}}_n^T) = \sum_{m=1}^M \sum_{n=1}^N p_{nm} \downarrow (\hat{\mathbf{y}}_m \cdot \hat{\mathbf{x}}_n^T)$$

where

$$\hat{\mathbf{y}}_m = \mathbf{y}_m - \mathbf{y}_r = \mathbf{y}_m - Y \cdot \mathbf{p}_r \quad \text{and} \quad \hat{Y} = Y \cdot (I_M - \hat{\mathbf{p}}_r \cdot \mathbf{1}_M^T) \quad (5.29)$$

Dropping out the vectorization, we subsequently have:

$$A \cdot \sum_{m=1}^M \sum_{n=1}^N p_{nm} (\hat{\mathbf{x}}_n \cdot \hat{\mathbf{x}}_n^T) = \sum_{m=1}^M \sum_{n=1}^N p_{nm} (\hat{\mathbf{y}}_m \cdot \hat{\mathbf{x}}_n^T)$$

This can be written in a matrix form as

$$A \cdot \hat{X} \cdot \text{dg}(\mathbf{p}_c) \cdot \hat{X}^T = \hat{Y} \cdot P^T \cdot \hat{X}^T$$

or simply

$$A \cdot \mathcal{R} = \mathcal{C} \implies A = \mathcal{C} \cdot \mathcal{R}^{-1} \quad (5.30)$$

where

$$\mathcal{R} = \hat{X} \cdot \text{dg}(\mathbf{p}_c) \cdot \hat{X}^T \quad \text{and} \quad \mathcal{C} = \hat{Y} \cdot P^T \cdot \hat{X}^T \quad (5.31)$$

are modified auto- and cross-correlation matrices, respectively. Both matrices are  $D \times D$ . The  $\mathcal{R}$  and  $\mathcal{C}$  matrices can be further modified using eqns (5.27), (5.29), (5.25) and (5.23) in the following way

$$\mathcal{R} = X \cdot P_C \cdot X^T \quad \text{and} \quad \mathcal{C} = Y \cdot \hat{P}^T \cdot X^T \quad (5.32)$$

where

$$P_C = \gamma \cdot \text{dg}(\mathbf{p}_c) \cdot \gamma^T, \quad \gamma = I_N - \frac{1}{\bar{p}} \mathbf{p}_c \cdot \mathbf{1}_N^T \quad (5.33)$$

and

$$\begin{aligned}
 \hat{P}^T &= (I_M - \hat{\mathbf{p}}_r \cdot \mathbf{1}_M^T) \cdot P^T \cdot (I_N - \mathbf{1}_N \cdot \hat{\mathbf{p}}_c^T) \\
 &= P^T - P^T \cdot \mathbf{1}_N \cdot \hat{\mathbf{p}}_c^T - \hat{\mathbf{p}}_r \cdot \mathbf{1}_M^T \cdot P^T + \hat{\mathbf{p}}_r \cdot \mathbf{1}_M^T \cdot P^T \cdot \mathbf{1}_N \cdot \hat{\mathbf{p}}_c^T \\
 &= P^T - \mathbf{p}_r \cdot \hat{\mathbf{p}}_c^T - \hat{\mathbf{p}}_r \cdot \hat{\mathbf{p}}_c^T + \hat{\mathbf{p}}_r \cdot \bar{p} \cdot \hat{\mathbf{p}}_c^T
 \end{aligned}$$

Finally we have

$$\hat{P}^T = P^T - \frac{1}{\bar{p}} \mathbf{p}_r \cdot \mathbf{p}_c^T \quad (5.34)$$

In summary, the point set registration with an affine transform can be described as follows:

### Initialization

- $\omega \in [0 \dots 1)$  — eqn (5.8)
- $Z = X$
- Calculate  $H$  — eqn (5.5) or (5.35)
- $\sigma^2 = \frac{1}{DNM} \mathbf{1}_N^T \cdot H \cdot \mathbf{1}_M$

Repeat until convergence ( $Q \leq Q_{min}$ )

### E-step

- Calculate  $H$  — eqn (5.5). In order to avoid a 3-D tensor induced by  $\mathbf{z}_n - \mathbf{y}_m$ , we can calculate the matrix  $H$  in the following way:

$$H = -2 \cdot Z^T \cdot Y +_c \text{dg}(Z^T \cdot Z) +_r \text{dg}(Y^T \cdot Y) \quad (5.35)$$

where  $Z^T \cdot Y$  is an  $N \times M$  matrix,  $\text{dg}(Z^T \cdot Z)$  and  $\text{dg}(Y^T \cdot Y)$  are  $N \times 1$  and  $1 \times M$  vectors that are added to all columns and vectors of the matrix  $Z^T \cdot Y$ , respectively  
 In MATLAB:

$$\mathbf{H} = -2 * \mathbf{Z}' * \mathbf{Y} + \text{diag}(\mathbf{Z}' * \mathbf{Z}) + \text{diag}(\mathbf{Y}' * \mathbf{Y})';$$

- Calculate  $E$  and  $P$  — eqn (5.17)
- Calculate  $\bar{p}$  — eqn (5.14),  $\mathbf{p}_c, \hat{\mathbf{p}}_c, \mathbf{p}_r$  — eqns (5.23), (5.25)
- Calculate  $P_C$  — eqn (5.33) and  $\hat{P}^T$  — eqn (5.34)

### M-step

- Calculate  $\mathcal{R}, \mathcal{C}$  — eqn (5.32) and  $A$  — eqn (5.30)
- Calculate  $\mathbf{b}$  — eqns (5.24), (5.25)
- Calculate the aligned point set,  $Z$  — eqns (5.26), (5.27) or eqn (5.22)
- Calculate the objective function,  $Q$  — eqns (5.18), (5.19)

## 5.4 Rigid point set registration

A rigid body transformation is similar to the affine transform of eqn (5.22) with the following restrictions:

$$A = sR, \quad R^T \cdot R = I_D, \quad \det(R) = +1 \quad (5.36)$$

where  $s$  is a scaling factor,  $R$  is a rotation matrix. A rotation matrix is an orthogonal matrix with the determinant equal to 1. the vector  $\mathbf{b}$  of eqn (5.22) is now a translation vector.

Following consideration of the previous section, the M-step optimal translation vector  $\mathbf{b}$  can be calculated as in eqn (5.24). To calculate the optimal value of the rotation matrix  $R$  given the conditions as in eqn (5.36) we first modify the objective function (5.13) by substituting (5.26). Ignoring terms independent of  $A$ , or  $R$ , we have:

$$\bar{Q} = \sum_{m=1}^M \sum_{n=1}^N p_{nm} (\mathbf{z}_n - \mathbf{y}_m)^T \cdot (\mathbf{z}_n - \mathbf{y}_m) = \sum_{m=1}^M \sum_{n=1}^N p_{nm} (A \cdot \hat{\mathbf{x}}_n - \hat{\mathbf{y}}_m)^T \cdot (A \cdot \hat{\mathbf{x}}_n - \hat{\mathbf{y}}_m)$$

where  $\hat{\mathbf{x}}_n$  and  $\hat{\mathbf{y}}_m$  are defined in eqns (5.27) and (5.29), respectively. Subsequently, we have:

$$\bar{Q} = \sum_{m=1}^M \sum_{n=1}^N p_{nm} (\hat{\mathbf{x}}_n^T \cdot A^T \cdot A \cdot \hat{\mathbf{x}}_n - 2\hat{\mathbf{y}}_m^T \cdot A \cdot \hat{\mathbf{x}}_n + \hat{\mathbf{y}}_m^T \cdot \hat{\mathbf{y}}_m)$$

From eqn (5.36) we note that  $A^T \cdot A = s^2 I_D$ , hence we further have:

$$\bar{Q} = \sum_{m=1}^M \sum_{n=1}^N p_{nm} (s^2 \hat{\mathbf{x}}_n^T \cdot \hat{\mathbf{x}}_n - 2s\hat{\mathbf{y}}_m^T \cdot R \cdot \hat{\mathbf{x}}_n + \hat{\mathbf{y}}_m^T \cdot \hat{\mathbf{y}}_m) \quad (5.37)$$

Using the concept of the trace of a matrix, we have the following equalities:

$$\begin{aligned} \sum_{m=1}^M \sum_{n=1}^N p_{nm} \hat{\mathbf{x}}_n^T \cdot \hat{\mathbf{x}}_n &= \text{tr}(X^T \cdot X \cdot \text{dg}(\mathbf{p}_c)) = \text{tr}(X \cdot \text{dg}(\mathbf{p}_c) \cdot X^T) \\ \sum_{m=1}^M \sum_{n=1}^N p_{nm} \hat{\mathbf{y}}_m^T \cdot \hat{\mathbf{y}}_m &= \text{tr}(Y \cdot \text{dg}(\mathbf{p}_r) \cdot Y^T) \\ \sum_{m=1}^M \sum_{n=1}^N p_{nm} \hat{\mathbf{y}}_m^T \cdot \hat{\mathbf{x}}_n &= \text{tr}(X \cdot P \cdot Y^T) \end{aligned} \quad (5.38)$$

Using these equalities, eqn (5.37) can be re-written in the following matrix form

$$\bar{Q} = s^2 \text{tr}(\hat{X} \cdot \text{dg}(\mathbf{p}_c) \cdot \hat{X}^T) - 2s \cdot \text{tr}(R \cdot \hat{X} \cdot P \cdot \hat{Y}^T) + \text{tr}(\hat{Y} \cdot \text{dg}(\mathbf{p}_r) \cdot \hat{Y}^T) \quad (5.39)$$

From this, we can calculate  $\frac{\partial \bar{Q}}{\partial s}$  and after equating it to zero we have the optimal scaling factor:

$$s = \text{tr}(R \cdot \hat{X} \cdot P \cdot \hat{Y}^T) / \text{tr}(\hat{X} \cdot \text{dg}(\mathbf{p}_c) \cdot \hat{X}^T) \quad (5.40)$$



We are ready to calculate the value of the rotation matrix  $R$  that minimises the modified objective function given by eqn (5.39). Since  $R$  is only in the middle term, minimization of  $\bar{Q}$  is equivalent to maximization of

$$\hat{Q} = \text{tr}(R \cdot \hat{R}^T) = \text{tr}(\hat{R}^T \cdot R) \quad \text{where} \quad \hat{R} = \hat{Y} \cdot P^T \cdot \hat{X}^T \quad (5.41)$$

Now we can write:

$$\hat{Q} = \sum_{d=1}^D \hat{\mathbf{r}}_d^T \cdot \mathbf{r}_d \quad (5.42)$$

where  $\hat{\mathbf{r}}_d$  and  $\mathbf{r}_d$  are column vectors of the matrices  $R$  and  $\hat{R}$ . It is clear that maximization of  $\hat{Q}$  can be achieved when vectors  $\hat{\mathbf{r}}_d$  and  $\mathbf{r}_d$  are collinear. The easiest way to achieve it is to make  $R = \hat{R}$  and impose the conditions on the rotation matrix as in eqn (5.36). One way of achieving this is to follow

Andriy Myronenko and Xubo Song, *On the Closed-Form Solution of the Rotation Matrix Arising in Computer Vision Problems*. arXiv:0904.1613v1, and perform the singular value decomposition of  $\hat{R}$ :

$$\hat{R} = \hat{Y} \cdot P^T \cdot \hat{X}^T = U \cdot S \cdot V^T \quad (5.43)$$

Then, the rotation matrix can use the same unitary matrices  $U$  and  $V$  with the singular values  $S$  being replaced by unities, namely:

$$R = V \cdot \xi \cdot U^T \quad \text{where} \quad \xi = \text{dg}([1, \dots, \det(V \cdot U^T)]) \quad (5.44)$$

where the last element of  $\xi$ ,  $\xi_D = \pm 1$  is selected such that  $\det R = +1$ .

It is interesting to know that in a 3D case ( $D = 3$ ) the rotation matrix has one eigenvalue equal to 1, and, typically a pair of complex conjugate eigenvalues that specifies the angle of rotation.

The overall rigid point set registration algorithm has the same initialization and the same E-step as for the affine registration. The M-step is appropriately adjusted to be:

### M-step

- Calculate  $\hat{R}$  and its *svd* — eqn (5.43)
- Calculate  $R$  — eqn (5.44)
- Calculate  $s$  — eqn (5.40)
- Calculate  $A$  — eqn (5.36)
- Calculate  $\mathbf{b}$  — eqns (5.24) , (5.25)
- Calculate the aligned point set,  $Z$  — eqns (5.26) , (5.27) or eqn (5.22)
- Calculate the objective function,  $Q$  — eqns (5.18) , (5.19)

## 5.5 Nonrigid point set registration — Coherent Point Drift Algorithm

This is an important case when two instantiations of an object are deformed, so that neither rigid body, nor affine transformation can describe such a deformation. It is a typical case for many medical images often taken in different modalities, e.g.  $X$  is from MRI, whereas  $Y$  are from ultrasonic images.

Following Tikhonov regularization framework

[https://en.wikipedia.org/wiki/Regularization\\_\(mathematics\)#Tikhonov\\_regularization](https://en.wikipedia.org/wiki/Regularization_(mathematics)#Tikhonov_regularization)  
the transformation is in the form of an initial position plus a displacement function  $v$ :

$$\mathbf{z}_n = \mathbf{x}_n + v(\mathbf{x}_n) \quad \text{or} \quad Z = X + v(X) \quad (5.45)$$

We add a regularization term to the objective function of eqn (5.13) to get:

$$Q(v, \sigma^2) = \frac{1}{2\sigma^2} \sum_{m=1}^M \sum_{n=1}^N p_{nm} h_{nm} + \frac{\bar{p}D}{2} \log \sigma^2 + \frac{\lambda}{2} \|Lv\|^2 \quad (5.46)$$

where  $L$  is differential regularization operator acting on the function  $v$  and  $\|Lv\|^2$  is the square of its norm. It is a well-known fact that the function  $v$  that minimizes the objective function 5.46 must satisfy the following Euler-Lagrange differential equation:

$$\frac{1}{\sigma^2 \lambda} \sum_{m=1}^M \sum_{n=1}^N p_{nm} (\mathbf{z}_n - \mathbf{y}_m) \delta(\mathbf{u} - \mathbf{x}_n) = \hat{L}Lv(\mathbf{u}) \quad (5.47)$$

for all vectors  $\mathbf{u}$ , where  $\hat{L}$  is the adjoint operator to  $L$ , and  $\mathbf{z}_n$  is given in eqn (5.45). The solution to such a differential equation is written in terms of a Green's function  $G$  of the self-adjoint operator  $\hat{L}L$  in the following form:

$$v(\mathbf{u}) = \frac{1}{\sigma^2 \lambda} \sum_{m=1}^M \sum_{n=1}^N p_{nm} (\mathbf{z}_n - \mathbf{y}_m) \cdot G(\mathbf{u}, \mathbf{x}_n) = \sum_{n=1}^N \mathbf{w}_n \cdot G(\mathbf{u}, \mathbf{x}_n) \quad (5.48)$$

where

$$\mathbf{w}_n = \frac{1}{\lambda \sigma^2} \sum_{m=1}^M p_{nm} (\mathbf{z}_n - \mathbf{y}_m) \quad (5.49)$$

where  $\mathbf{w}_n$  is a  $D \times 1$  vector of coefficients being an  $n$ th column of a respective matrix  $W$ . The Green's function can be selected to be a matrix of Gaussians with the entries:

$$g_{k,n} = G(\mathbf{x}_k, \mathbf{x}_n) = \exp\left(-\frac{\|\mathbf{x}_k - \mathbf{x}_n\|^2}{2\beta^2}\right) \quad (5.50)$$

This, following eqn (5.35), can be calculated as

$$\mathcal{X} = X^T \cdot X, \quad \hat{G} = -2\mathcal{X} +_c \text{dg}(\mathcal{X}) +_r \text{dg}(\mathcal{X}), \quad G = \exp\left(-\frac{\mathcal{X}}{2\beta^2}\right) \quad (5.51)$$

Re-writing eqn (5.48) in the matrix form we have:

$$v(X) = W \cdot G \quad \text{hence} \quad Z = X + W \cdot G \quad (5.52)$$

The matrix of coefficients from eqn (5.49) can be further expressed as:

$$W = \frac{1}{\lambda\sigma^2}(Z \cdot \text{dg}(\mathbf{p}_c) - Y \cdot P^T)$$

Expanding  $Z$  as in (5.52) gives:

$$(X + W \cdot G) \cdot \text{dg}(\mathbf{p}_c) - Y \cdot P^T = \lambda\sigma^2 W$$

Re-arranging the terms, we can get an equation for the matrix of coefficients  $W$ :

$$W \cdot (G - \lambda\sigma^2 \text{dg}(\mathbf{p}_c)^{-1}) = Y \cdot P^T \cdot \text{dg}(\mathbf{p}_c)^{-1} - X \quad (5.53)$$

The optimal value of  $\sigma^2$  is given by eqns (5.20) and (5.19), namely

$$\sigma^2 = \frac{1}{\bar{p}D} \text{tr}(H \cdot P^T) \quad (5.54)$$

where  $H$  is specified in eqn (5.5). We also have:

$$\text{tr}(H \cdot P^T) = \text{tr}(Y \cdot \text{dg}(\mathbf{p}_r) \cdot Y^T) + \text{tr}(Z \cdot \text{dg}(\mathbf{p}_c) \cdot Z^T) - 2 \text{tr}(Z \cdot P \cdot Y^T) \quad (5.55)$$

The overall non-rigid point set registration algorithm has the similar initialization. This time there are three parameters:  $\omega, \beta, \lambda$

- Initialize:  $\omega \in [0 \dots 1)$  — eqn (5.8),  $\beta > 0$ ,  $\lambda > 0$
- Initialize:  $Z = X$ ,  $W = 0$
- Calculate  $H$  — eqn (5.35)
- Calculate:  $\sigma^2 = \frac{1}{DNM} \mathbf{1}_N^T \cdot H \cdot \mathbf{1}_M$  — eqn (5.5)
- Calculate  $G$  — eqn (5.51)

and the same E-step as for the affine registration. The M-step is appropriately adjusted to be:

### M-step

- Solve eqn (5.53) for  $W$
- The aligned point set  $Z$  is given by eqn (5.52)
- Calculate  $\sigma^2$  — eqn (5.54)
- Calculate the objective function,  $Q$  — eqns (5.18), (5.19)

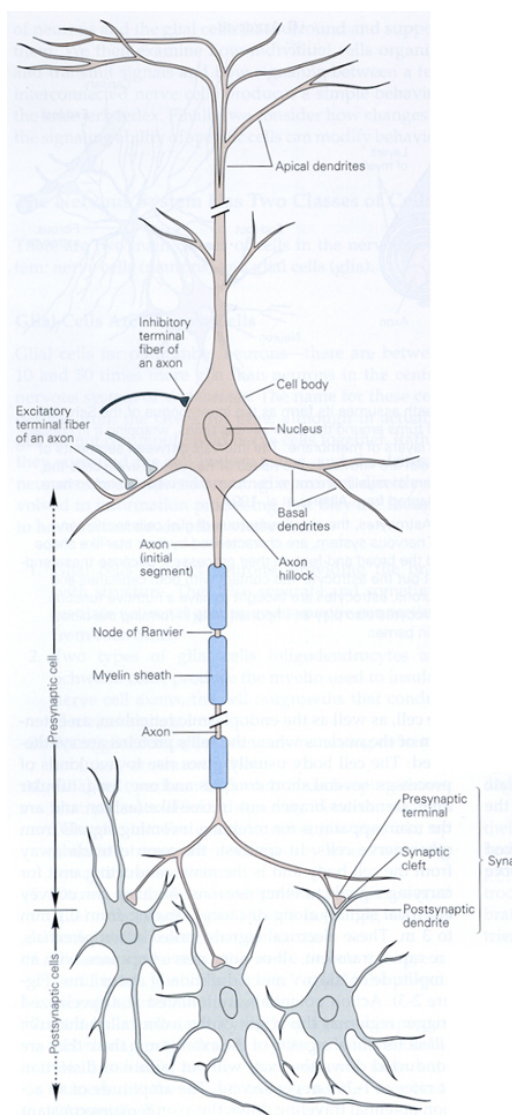
## 6 Structure of Neural Networks

Neural Networks are non-linear mapping devices/algorithms, therefore, its mathematical foundations are covered in sections 1, 2 and 3.

### 6.1 Biological Foundations of Neural Networks

This section is non-essential wrt mathematical aspects of artificial neural networks and can be omitted in the first reading.

We start with considering the structure of a typical biological neuron, or a nerve cell. Be aware of imprecision behind the word “typical”.



(From Kandel, Schwartz and Jessel, *Principles of Neural Science*)

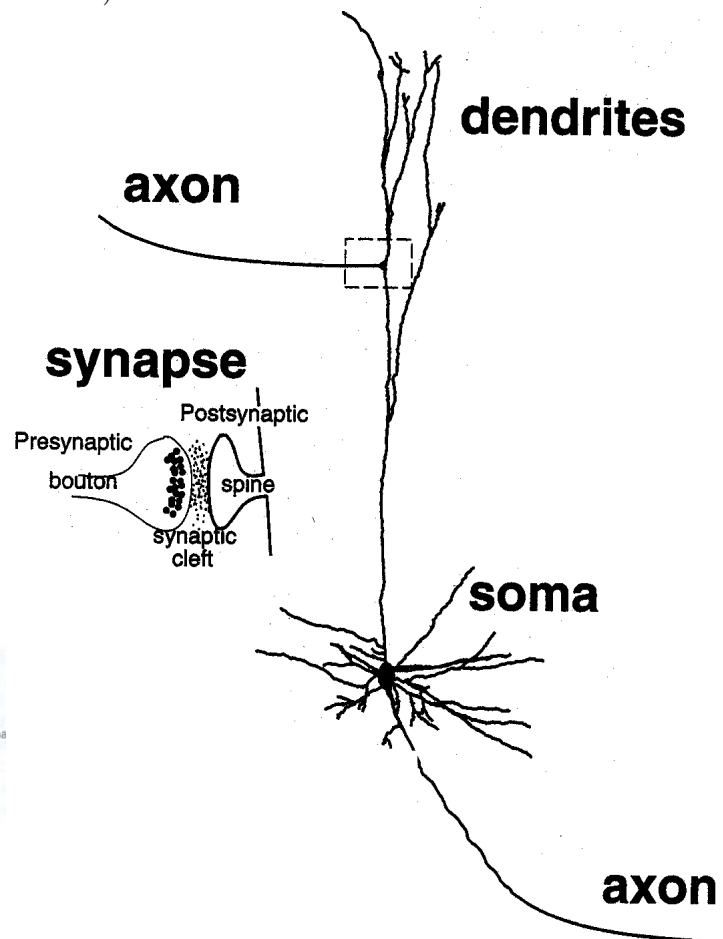
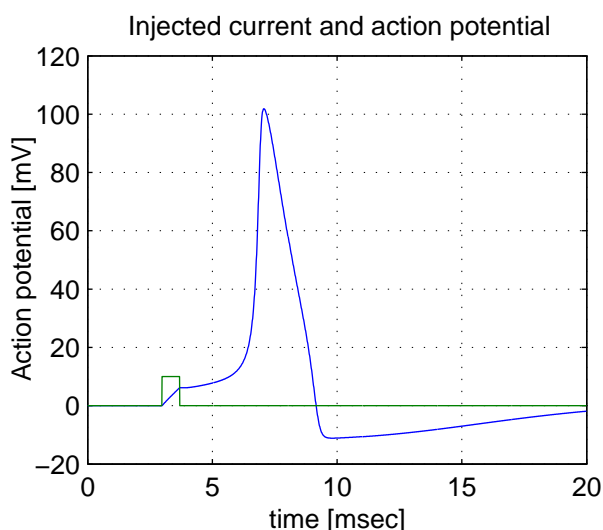


Figure 6–1: Structures of “typical” biological neurons.

- The **cell body** contains the nucleus, the storehouse of genetic information, and gives rise to two types of cell processes, **axons** and **dendrites**.
- Axons, the output transmitting element of neurons, can vary greatly in length; some can extend more than 3m within the body. Most axons in the central nervous system are very thin ( $0.2 \dots 20 \mu\text{m}$  in diameter) compared with the diameter of the cell body ( $50 \mu\text{m}$  or more).
- Many axons are insulated by a fatty sheath of myelin that is interrupted at regular intervals by the nodes of Ranvier.
- The **action potential**, the cell's conducting signal, is initiated either at the **axon hillock**, the initial segment of the axon, or in some cases slightly farther down the axon at the first nod of Ranvier. The single action potential is similar to the following:



- Branches of the axon of one neuron (the presynaptic neuron) transmit signals to another neuron (the postsynaptic cell) at a site called the **synapse**.
- The branches of a single axon may form synapses with as many as 1000 other neurons.
- Whereas the axon is the output element of the neuron, the **dendrites** (apical and basal) are input elements of the neuron. Together with the cell body, they receive synaptic signals from other neurons.

Simplified functions of these very complex in their nature “building blocks” of a neuron are as follow:

- The synapses are elementary signal processing devices.

- A synapse is a biochemical device which converts a pre-synaptic electrical signal into a chemical signal and then back into a post-synaptic electrical signal.
  - In the synapse, neuro-transmitters (information-carrying chemicals) are released pre-synaptically, floats across the synaptic cleft, and activate receptors postsynaptically.
  - The input pulse train has its amplitude modified by parameters stored in the synapse. The nature of this modification depends on the type of the synapse, which can be either inhibitory or excitatory.
  - The postsynaptic signals are aggregated and transferred along the dendrites to the nerve cell body.
  - The cell body generates the output neuronal signal, activation potential, which is transferred along the axon to the synaptic terminals of other neurons.
  - The frequency of firing of a neuron is proportional to the total synaptic activities and is controlled by the synaptic parameters (weights).
  - The pyramidal cell can receive  $10^4$  synaptic inputs and it can fan-out the output signal to thousands of target cells — the connectivity difficult to achieve in the artificial neural networks.
- According to Calaj’s “neuron-doctrine” information carrying signals come into the dendrites through synapses, travel to the cell body, and activate the axon. Axonal signals are then supplied to synapses of other neurons.

## 6.2 A simplistic model of a biological neuron

Basic signal processing characteristics of a biological neuron:

- data is coded in a form of instantaneous frequency of pulses
- synapses are either excitatory or inhibitory
- Signals are aggregated (“summed”) when travel along dendritic trees
- The cell body (neuron output) generates the output pulse train of an average frequency proportional to the total (aggregated) post-synaptic activity (activation potential).

The brain is a highly complex, non-linear, parallel information processing system. It performs tasks like pattern recognition, perception, motor control, many times faster than the fastest digital computers.

- Biological neurons, the basic building blocks of the brain, are slower than silicon logic gates. The neurons operate in milliseconds which is about six–seven orders of magnitude slower than the silicon gates operating in the sub-nanosecond range.
- The brain makes up for the slow rate of operation with two factors:

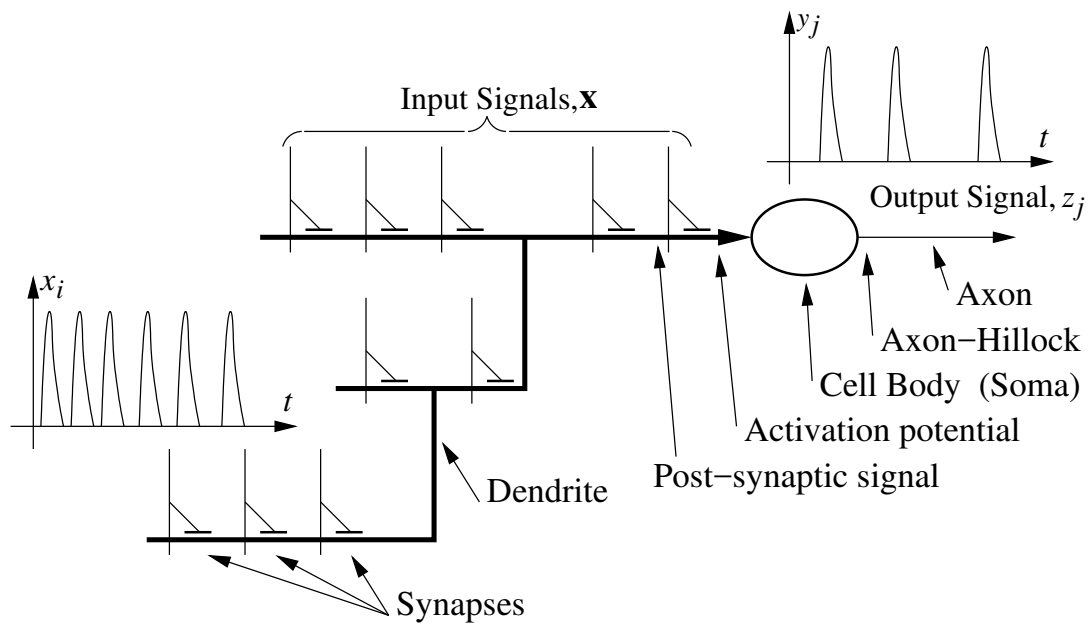


Figure 6–2: Conceptual structure of a biological neuron

- a huge number of nerve cells (neurons) and interconnections between them. The number of neurons is estimated to be in the range of  $10^{10}$  with  $60 \cdot 10^{12}$  synapses (interconnections).
- A function of a biological neuron seems to be much more complex than that of a logic gate.
- The brain is very energy efficient. It consumes only about  $10^{-16}$  joules per operation per second, comparing with  $10^{-6}$  J/oper·sec for a digital computer.

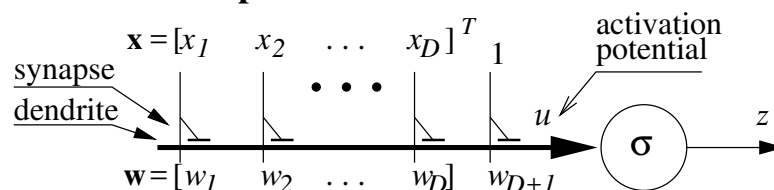
### 6.3 Models of artificial neurons

Artificial neural networks are nonlinear information (signal) processing devices which are built from interconnected elementary processing devices called neurons.

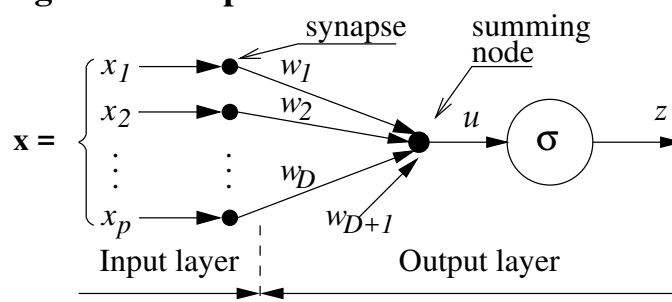
An artificial neuron is a  $D$ -input single-output signal processing element which can be thought of as a simple model of a non-branching biological neuron.

Three basic graphical representations of a single  $D$ -input ( $D$ -synapse) neuron:

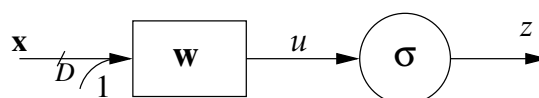
#### a. Dendritic representation



#### b. Signal flow representation



#### c. Block-diagram representation



$$u = w_1 \cdot x_1 + \dots + w_D \cdot x_D + w_{D+1} = \mathbf{w} \cdot \hat{\mathbf{x}}, \quad \hat{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}$$

$$z = \sigma(u)$$

where  $\hat{\mathbf{x}}$  is the augmented input vector.

From a **dendritic representation** of a single **neuron** we can identify  $D+1$  **synapses** arranged along a linear **dendrite** which aggregates the synaptic activities, and a neuron body or axon-hillock generating an output signal.

The **pre-synaptic activities** are represented by a  $D$ -element **column vector** of input (afferent) signals

$$\mathbf{x} = [x_1 \dots x_D]^T, \quad \hat{\mathbf{x}} = [x_1 \dots x_D \ 1]^T$$



Hence, the space of input patterns is  $D$ -dimensional. In addition, there is a constant **biasing** input equal to 1.

Synapses are characterised by adjustable parameters called weights or synaptic strength parameters. The weights are arranged in a  $D + 1$ -element **row vector**:

$$\mathbf{w} = [w_1 \ \dots \ w_D \ w_{D+1}]$$

The parameter  $w_{D+1}$  specifies the **bias**.

- In a **signal flow** representation of a neuron,  $D + 1$  synapses are arranged in a layer of input **nodes**. A dendrite is replaced by a single summing node. Weights are now attributed to **branches** (connections) between input nodes and the summing node.
- Passing through synapses and a dendrite (or a summing node), input signals are aggregated (combined) into the **activation potential**,  $u$ , which describes the total **post-synaptic activity**.
- The activation potential is formed as a linear combination of input signals and synaptic strength parameters, that is, as an **inner product** of the weight and input vectors:

$$u = \sum_{i=1}^D w_i x_i + w_{D+1} = \mathbf{w} \cdot \hat{\mathbf{x}} = \begin{bmatrix} w_1 & w_2 & \dots & w_{D+1} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \\ 1 \end{bmatrix} \quad (6.1)$$

- Subsequently, the activation potential (the total post-synaptic activity) is passed through an **activation function**,  $\sigma(\cdot)$ , which generates the output (efferent) signal:

$$z = \sigma(u) \quad (6.2)$$

- The activation function is typically a saturating function which normalises the total post-synaptic activity to the standard values of output (axonal) signal. Many other options are possible
- The **block-diagram** representation encapsulates basic operations of an artificial neuron, namely, aggregation of pre-synaptic activities, eqn (6.1), and generation of the output signal, eqn (6.2).

## 6.4 Types of activation functions

You can skip detail of this section in first reading and go to sec. 6.5. Activation functions become important in the learning procedures.

Typically, the activation function

$$z = \sigma(u)$$

generates either **unipolar** or **bipolar** signals.

Many learning algorithms also require calculation of the derivative of the activation function

$$z' = \frac{d\sigma}{du}$$

**A linear function:**  $z = u$ .

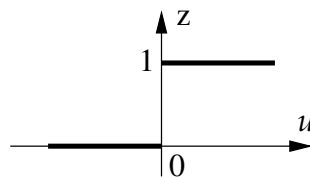
Such linear processing elements, sometimes called ADALINEs, are studied in the theory of linear systems, for example, in the “traditional” signal processing and statistical regression analysis.

Note that  $\frac{dz}{du} = 1$ .

**A step function**

unipolar:

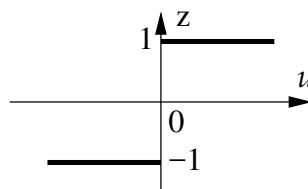
$$z = \sigma(u) = \begin{cases} 1 & \text{if } u \geq 0 \\ 0 & \text{if } u < 0 \end{cases}$$



Such a processing element is traditionally called **perceptron**, and it works as a threshold element with a binary output.

bipolar:

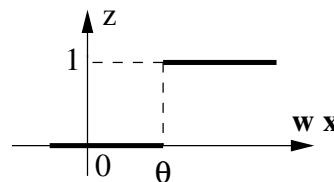
$$z = \sigma(u) = \begin{cases} +1 & \text{if } u \geq 0 \\ -1 & \text{if } u < 0 \end{cases}$$



**A step function with bias**

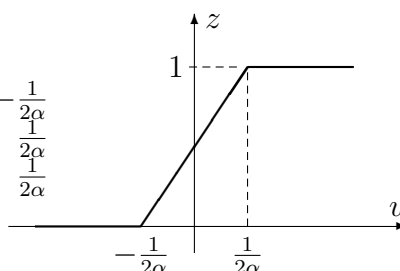
The bias (threshold) can be added to both, unipolar and bipolar step function. We then say that a neuron is “fired”, when the synaptic activity exceeds the threshold level,  $\theta$ . For a unipolar case, we have:

$$z = \sigma(u) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} \geq \theta \\ 0 & \text{if } \mathbf{w} \cdot \mathbf{x} < \theta \end{cases}$$



Note that the derivative of the step function is zero apart from the point of discontinuity. (The McCulloch-Pitts perceptron — 1943)

### A piecewise-linear function

$$z = \sigma(u) = \begin{cases} 0 & \text{if } u \leq -\frac{1}{2\alpha} \\ \alpha u + \frac{1}{2} & \text{if } |u| < \frac{1}{2\alpha} \\ 1 & \text{if } u \geq \frac{1}{2\alpha} \end{cases}$$


- For small activation potential,  $u$ , the neuron works as a linear combiner (an ADA-LINE) with the gain (slope)  $\alpha$ .
- For large activation potential,  $v$ , the neuron saturates and generates the output signal either 0 or 1.
- For large gains  $\alpha \rightarrow \infty$ , the piecewise-linear function is reduced to a step function.
- The derivative is equal to  $\alpha$  in the linear part and zero otherwise.

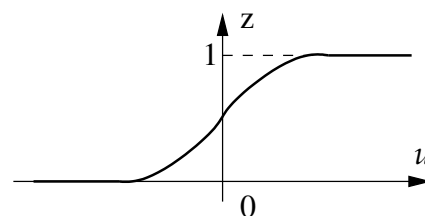
### Logistic sigmoid functions

unipolar:

$$z = \sigma(u) = \frac{1}{1 + e^{-u}} = \frac{1}{2}(\tanh(u/2) + 1)$$

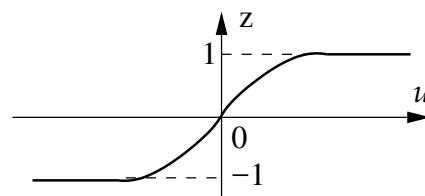
The derivative:

$$\sigma' = \frac{dz}{du} = z(1 - z), \quad \sigma'(0) = 1$$



bipolar:

$$z = \sigma(u) = \tanh(\beta u) = \frac{2}{1 + e^{-2\beta u}} - 1$$



The parameter  $\beta$  controls the slope of the function.

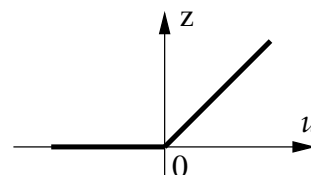
The hyperbolic tangent (bipolar sigmoid) function is perhaps the most popular choice of the activation function specifically in problems related to function mapping and approximation.

The derivative:  $\sigma' = \frac{dz}{du} = \beta(1 - z^2)$   $\sigma'(0) = \beta$

### ReLU – Rectifier function (ramp function)

$$z = \sigma(u) = \max(0, u) = \begin{cases} u & \text{if } u \geq 0 \\ 0 & \text{if } u < 0 \end{cases}$$

The derivative:  $\sigma' = \begin{cases} 1 & \text{if } u \geq 0 \\ 0 & \text{if } u < 0 \end{cases}$



### Radial-Basis Functions

- Radial-basis functions arise as optimal solutions to problems of interpolation, approximation and regularisation of functions. The optimal solutions to the above problems are specified by some integro-differential equations which are satisfied by a wide range of nonlinear differentiable functions (S. Haykin, *Neural Networks – a Comprehensive Foundation*, Ch. 5).
- Typically, Radial-Basis Functions  $\varphi(\mathbf{x}; \mathbf{t}_i)$  form a family of functions of a  $D$ -dimensional vector,  $\mathbf{x}$ , each function being centered at point  $\mathbf{t}_i$ .
- A popular simple example of a Radial-Basis Function is a symmetrical multivariate Gaussian function which depends only on the distance between the current point,  $\mathbf{x}$ , and the center point,  $\mathbf{t}_i$ , and the variance parameter  $\sigma_i$ :

$$\varphi(\mathbf{x}; \mathbf{t}_i) = G(\|\mathbf{x} - \mathbf{t}_i\|) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{t}_i\|^2}{2\sigma_i^2}\right)$$

where  $\|\mathbf{x} - \mathbf{t}_i\|$  is the norm of the distance vector between the current vector  $\mathbf{x}$  and the centre,  $\mathbf{t}_i$ , of the symmetrical multidimensional Gaussian surface.

- The spread of the Gaussian surface is controlled by the variance parameter  $\sigma_i$ .

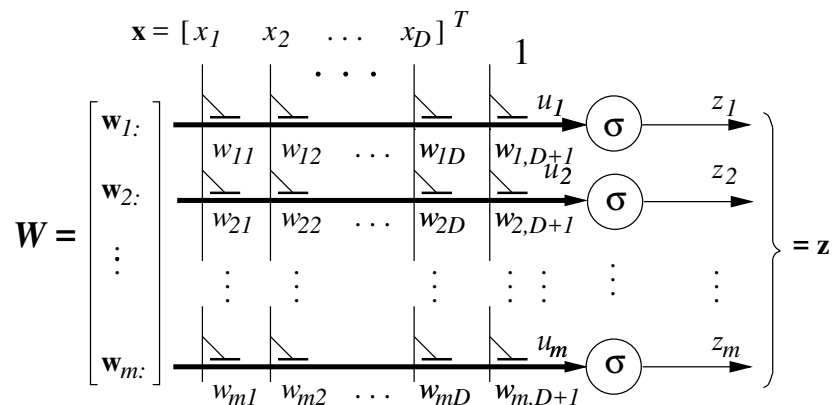
## 6.5 A layer of neurons

Neurons as in sec. 6.3 can be arranged into a **layer** of neurons.

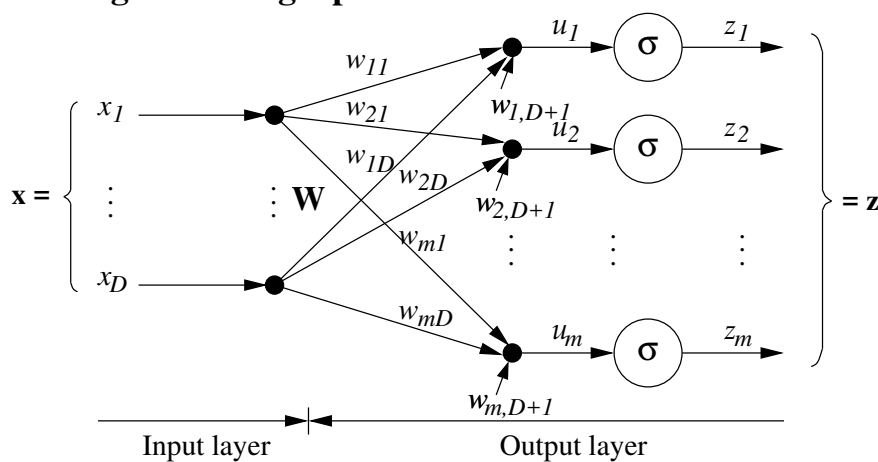
A **single layer neural network** consists of  $m$  neurons each with the same  $D$  input signals plus a constant **bias input**

Similarly to a single neuron, the neural network can be represented in all **three** basic forms: **dendritic**, **signal-flow**, and **block-diagram** form:

### a. Dendritic graph



### b. Signal-flow graph



### c. Block-diagram

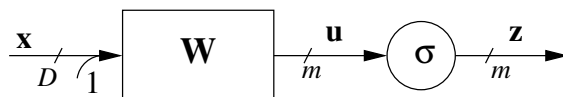


Figure 6-3: Three representations of a single layer neural network

- From the **dendritic representation** of the neural network it is readily seen that a layer of neurons is described by a  $m \times (D + 1)$  matrix  $W$  of synaptic weights.

- Each row of the **weight matrix** is associated with one neuron.

Operations performed by the network can be described as follows:

$$\underbrace{\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_m \end{bmatrix}}_{\mathbf{u}} = \underbrace{\begin{bmatrix} w_{11} & \cdots & w_{1D} & w_{1D+1} \\ w_{21} & \cdots & w_{2D} & w_{2D+1} \\ \vdots & \cdots & \vdots & \vdots \\ w_{m1} & \cdots & w_{mD} & w_{mD+1} \end{bmatrix}}_W \underbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_D \\ 1 \end{bmatrix}}_{\hat{\mathbf{x}}}; \quad \underbrace{\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix}}_{\mathbf{z}} = \underbrace{\begin{bmatrix} \sigma(u_1) \\ \sigma(u_2) \\ \vdots \\ \sigma(u_m) \end{bmatrix}}_{\boldsymbol{\sigma}(\mathbf{u})}$$

or in a matrix form as:

$$\hat{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}, \quad \mathbf{u} = W \cdot \hat{\mathbf{x}}, \quad \mathbf{z} = \boldsymbol{\sigma}(\mathbf{u}) = \boldsymbol{\sigma}(W \cdot \hat{\mathbf{x}}) \quad (6.3)$$

where  $\mathbf{u}$  is a vector of activation potentials. The calculation steps are as follows:

- The vector of input signals  $\mathbf{x}$  is augmented with a bias constant input signal “1”
- The vector  $\mathbf{u}$  of activating potential aka output from the linear part of the network is calculated
- $\mathbf{u}$  is passed through the nonlinear part of the system  $\boldsymbol{\sigma}$  to create the vector of the output signals  $\mathbf{z}$

It is sometime convenient to explicitly split the weight matrix into its “signal” and bias part as follows:

$$W = [\bar{W} \quad \mathbf{b}]; \quad \bar{W} = \begin{bmatrix} w_{11} & \cdots & w_{1D} \\ w_{21} & \cdots & w_{2D} \\ \vdots & \cdots & \vdots \\ w_{m1} & \cdots & w_{mD} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} w_{1D+1} \\ w_{2D+1} \\ \vdots \\ w_{mD+1} \end{bmatrix} \quad (6.4)$$

so that

$$\mathbf{u} = W \cdot \hat{\mathbf{x}} = \bar{W} \cdot \mathbf{x} + \mathbf{b} \quad (6.5)$$

From the **signal-flow graph** it is visible that each weight parameter  $w_{ij}$  (synaptic strength) is now related to a connection between nodes of the input layer and the output layer.

Therefore, the name **connection strengths** for the weights is also justifiable.

The **block-diagram** representation of the single layer neural network is the most compact one, hence most convenient to use.

## 6.6 Feedforward Multilayer aka Deep Neural Networks

Feedforward multilayer neural networks are one of the major tools used in machine learning. Pattern recognition and classification being the major tasks. Such a network is trained for a set of patterns and then it is expected to recognize an unknown pattern. During the training, or learning step the correct weight parameters are being derived.

A multilayer neural network consists of  $K$  layers of neurons as presented in sec. 6.5. The  $k$ th layer is described by the following equations:

$$\begin{aligned} \mathbf{x}_n^{(k)} &= \mathbf{z}_n^{(k-1)} \quad , \quad \hat{\mathbf{x}}_n^{(k)} = \begin{bmatrix} \mathbf{x}_n^{(k)} \\ 1 \end{bmatrix} \quad \text{for } n = 1, \dots, N \quad \text{and} \quad \text{for } k = 1, \dots, K \\ \mathbf{u}_n^{(k)} &= W^{(k)} \cdot \hat{\mathbf{x}}_n^{(k)} \\ \mathbf{z}_n^{(k)} &= \sigma(\mathbf{u}_n^{(k)}) \end{aligned} \quad (6.6)$$

$\mathbf{x}_n^{(1)} = \mathbf{x}_n$  — input to the first layer ;  $\mathbf{z}_n^{(K)} = \mathbf{z}_n$  — output from the last layer

Descriptively we can say that for each  $n$ th input pattern/stimulus and for each layer  $k$ th

- An input to a  $k$ th layer  $\mathbf{x}_n^{(k)}$  is formed from the output from the  $(k-1)$ th layer  $\mathbf{z}_n^{(k-1)}$
- A biasing constant input is added to create an augmented input vector  $\hat{\mathbf{x}}_n^{(k)}$
- the postsynaptic activity  $\mathbf{u}_n^{(k)}$  is created using the weight matrix  $W^{(k)}$
- The output from the layer is created passing  $\mathbf{u}_n^{(k)}$  through the nonlinearity  $\sigma$

A network of  $K$  layers as in eqn (6.6) is known as **Multilayer Perceptrons (MLP)**, or **Multilayer Neural Network (nnet)** and more recently as a **deep neural network**.

It is easy to identify that eqn (6.6) is a special case of a general nonlinear mapping described in eqn (1.2) and elaborated on in sec. 3.

## 6.7 Two-layer neural network

The smallest “deep” neural network consists of **two layers of neurons** as in the following block-diagram:

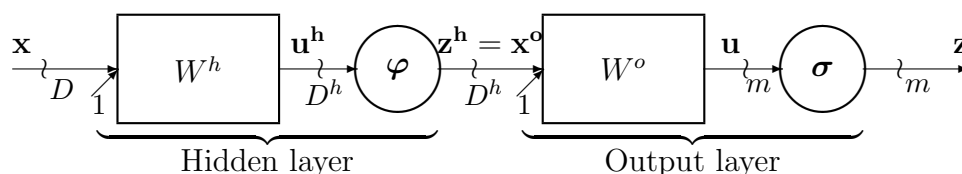


Figure 6-4: Block diagram of a two-layer neural network

Note:

- $D$  input signals,  $\mathbf{x}$ , and biasing input “1”
- $D^h$  hidden neurons generating signals  $\mathbf{u}^h$  and  $\mathbf{z}^h$ . For simplicity we have omitted the parenthesis in the superscripts as in  $D^{(h)}$
- $m$  output neurons generating signals  $\mathbf{u}$  and  $\mathbf{z}$ ,
- biasing inputs at both layers.

The two-layer architecture is often referred to as a **single hidden layer** neural network.

Input signals,  $\mathbf{x}$ , are passed through synapses of the hidden layer with connection strengths described by the **hidden weight matrix**,  $W^h$ , and the  $D^h$  **hidden activation signals**,  $\mathbf{u}^h$ , are generated.

The hidden activation signals are then normalised by the functions  $\psi$  into the  $D^h$  **hidden signals**,  $\mathbf{z}^h$  that form the inputs  $\mathbf{x}^o$  to the output layer

Similarly, the input signals to the output layer  $\mathbf{x}^o = \mathbf{z}^h$ , are first, converted into  $m$  **output activation signals**,  $\mathbf{u}^o$ , by means of the output weight matrix,  $W^o$ , and subsequently, into  $m$  **output signals**,  $\mathbf{z}$ , by means of the functions  $\sigma$ . Hence

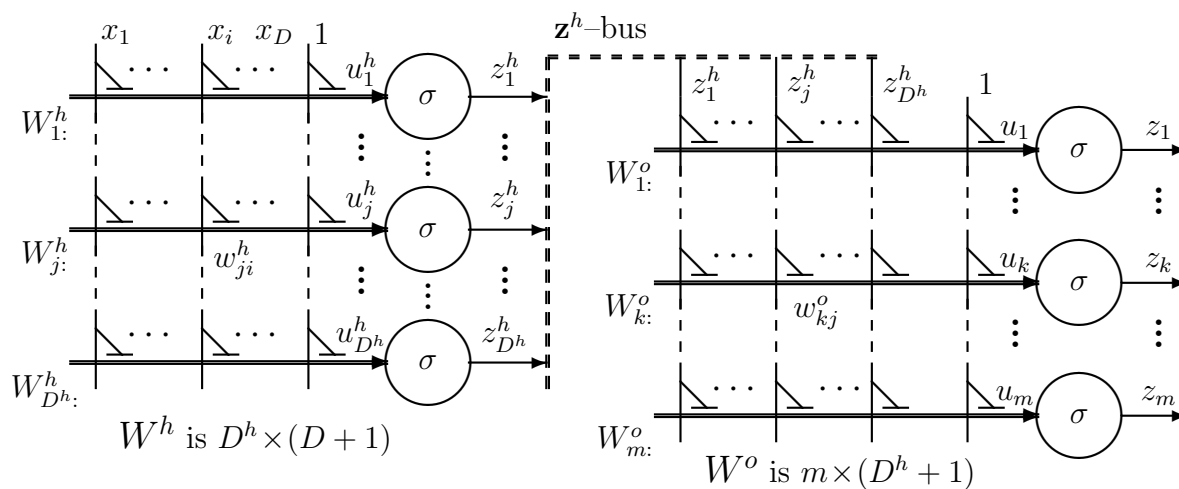
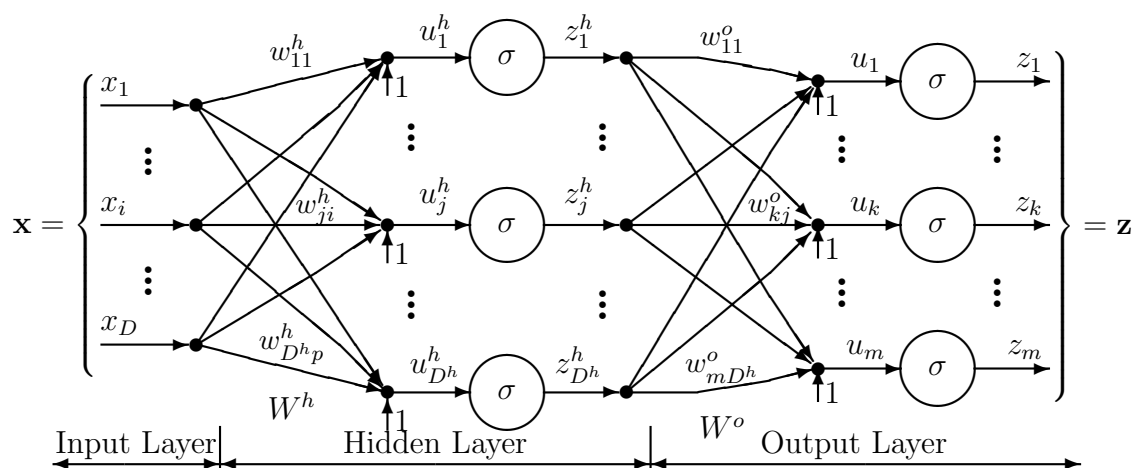
$$\hat{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix}; \quad \mathbf{z}^h = \varphi(W^h \cdot \hat{\mathbf{x}}), \quad \hat{\mathbf{x}}^o = \begin{bmatrix} \mathbf{z}^h \\ 1 \end{bmatrix}, \quad \mathbf{z} = \sigma(W^o \cdot \hat{\mathbf{x}}^o) \quad (6.7)$$

Functions  $\varphi$  and  $\sigma$  can be identical.

The block-diagram representation of a two-layer neural network as in Fig. 6–4 can also be presented in a dendritic and data-flow forms, similarly to a single layer of neurons presented in Fig. 6–3.

Note the constant unity inputs added to each layer that form the biasing inputs. The nonlinear functions  $\sigma$  in two layers can be, and often are, different.



**Dendritic diagram:****Signal-flow diagram:**

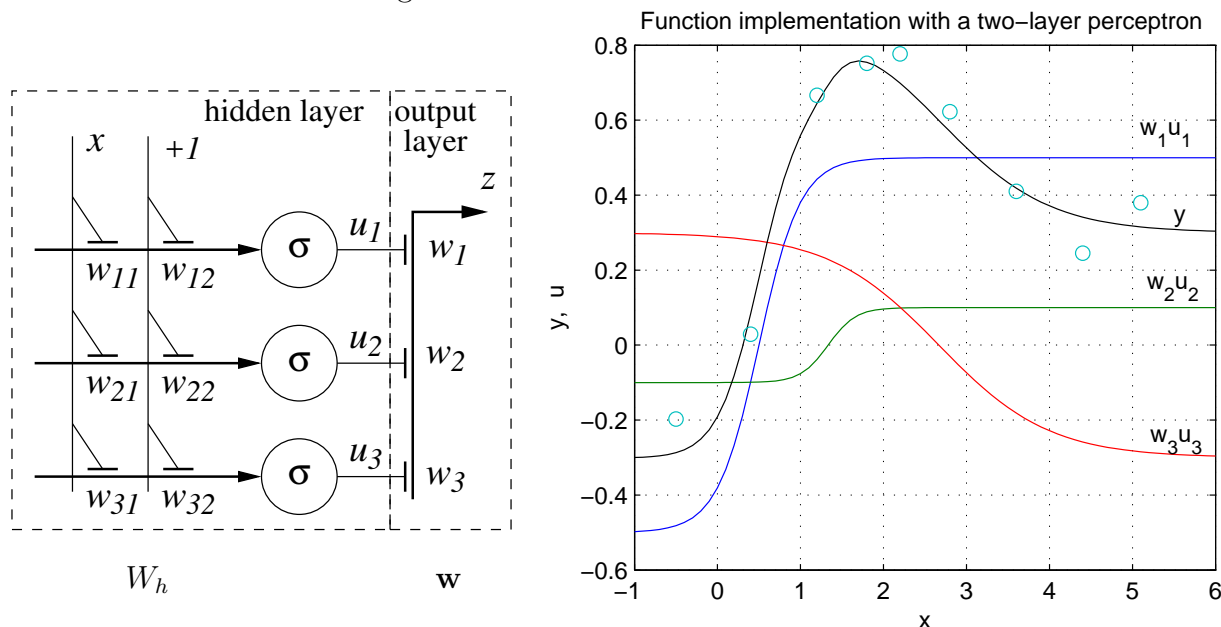
$$u_j^h = W_j^h \cdot \hat{\mathbf{x}}; \quad z_j^h = \sigma(u_j^h); \quad u_k = W_k^o \cdot \hat{\mathbf{z}}^h; \quad z_k = \sigma(u_k)$$

$$\mathbf{z}^h = \sigma(W^h \cdot \hat{\mathbf{x}}), \quad \mathbf{z} = \sigma(W^o \cdot \hat{\mathbf{z}}^h)$$

Figure 6-5: Two representations of a two-layer neural network

## 6.8 Example of a function implemented by a two-layer nnet

Consider a single-variable function  $y = f(x)$  implemented by the following two-layer nnet. Note that we assume that the weight matrices are known.



The first layer is described by the  $3 \times 2$  matrix  $W^h$ :

$$W^h = \begin{bmatrix} w_{11}^h & w_{12}^h \\ w_{21}^h & w_{22}^h \\ w_{31}^h & w_{32}^h \end{bmatrix} = \begin{bmatrix} 2 & -1 \\ 3 & -4 \\ 0.75 & -2 \end{bmatrix}$$

that is followed by the function  $\sigma = \tanh(\mathbf{u})$ .

The second layer's weight matrix is reduced to a vector:

$$\mathbf{w} = [w_1 \ w_2 \ w_3] = [0.5 \ 0.1 \ -0.3]$$

The neural net implements the following function:

$$\begin{aligned} z &= \mathbf{w} \cdot \mathbf{u} = \mathbf{w} \cdot \tanh \left( W^h \cdot \begin{bmatrix} x \\ 1 \end{bmatrix} \right) = w_1 \cdot u_1 + w_2 \cdot u_2 + w_3 \cdot u_3 \\ &= w_1 \cdot \tanh(w_{11}^h \cdot x + w_{12}^h) + w_2 \cdot \tanh(w_{21}^h \cdot x + w_{22}^h) + w_3 \cdot \tanh(w_{31}^h \cdot x + w_{32}^h) \\ &= 0.5 \cdot \tanh(2x - 1) + 0.1 \cdot \tanh(3x - 4) - 0.3 \cdot \tanh(0.75x - 2) \end{aligned}$$

## 7 Learning Algorithms for Feedforward/Deep Neural Networks

In this section we will start our journey through the fundamentals of the supervising learning algorithms. We first start with the error-correcting algorithms and introduce the concept of error back-propagation

### 7.1 Fundamentals of Error-Correcting Learning Algorithms

Error-correcting learning algorithms are supervised learning algorithms that modify the parameters  $\mathbf{w}$  of the network in such a way to minimise that error between the desired  $y$  and actual outputs  $z$  as illustrated in Fig. 7-1

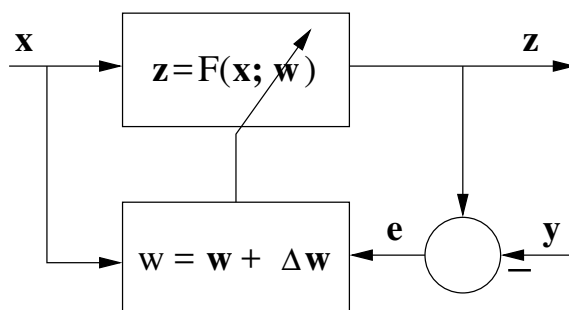


Figure 7-1: The principle of error-correcting learning

Repeating the considerations of sec. 3 we can say that learning is a two step procedure

- in the feedforward pass, for a current input pattern  $\mathbf{x}_n$ , we calculate the output of the network

$$\mathbf{z}_n = F(\mathbf{x}_n; \mathbf{w})$$

- In the feedback pass, we first calculate the point aka pattern error vector

$$\mathbf{e}_n = \mathbf{z}_n - \mathbf{y}_n = F(\mathbf{x}_n; \mathbf{w}) - \mathbf{y}_n \quad (7.1)$$

then the pattern squared error function

$$E_n(\mathbf{w}) = \frac{1}{2} \mathbf{e}_n^T \cdot \mathbf{e}_n \quad (7.2)$$

next, the total mean squared error function, aka the performance function:

$$E(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^N E_n(\mathbf{w}) \quad (7.3)$$

(compare with eqn (1.5)).

After that we calculate the gradient of the performance function  $\nabla E(\mathbf{w})$  and additively update the vector of parameters  $\mathbf{w}$  using any method described in sec. 3, the **gradient decent method** being the simplest, hence the most popular:

$$\Delta \mathbf{w} = -\eta \nabla E(\mathbf{w})$$

where  $\eta$  is the learning rate.

The vector of parameters  $\mathbf{w}$  includes the elements of ALL weight matrices. In the case of the multilayer nnet that would give:

$$\mathbf{w} = \text{vec}([W_1 \ W_2 \ \dots \ W_K]) \quad (7.4)$$

In the next section we will study a method of updating each weight matrix  $W_k$ , layer-by-layer using the concept of error backpropagation:

$$W^{(k)} = W^{(k)} + \Delta W^{(k)} \quad \text{for each layer } k = K, K-1, \dots, 1$$

As an introductory comment, we note that the weights can be updated in two ways:

- in the **pattern** mode weights are updated after the presentation of each learning pattern  $\mathbf{x}_n$ , using the pattern performance function and the resulting pattern gradient

$$\Delta \mathbf{w}_n = -\eta \nabla E_n(\mathbf{w}) \quad (7.5)$$

- in the **batch** mode weights are updated after each epoch, that is presentation of all  $N$  datapoints  $X$ , using the total performance function and the resulting total gradient

$$\Delta \mathbf{w} = -\eta \nabla E(\mathbf{w}) \quad (7.6)$$

Finally, we need to calculate the gradient, or pattern gradient, for a specific form of nonlinearity as used in nnets and described in sec. 6.6, eqn (6.6). Descriptively, a nnet consists of  $K$  layers, each having the linear mapping with the weight matrix  $W^k$  followed by the nonlinearity  $\sigma(\cdot)$ .

## 7.2 Backpropagation of Errors in Two-layer nnets

In this section we will calculate gradients for layers of the neural network specified in eqn (6.6). For simplicity we start with a two-layer network as presented in Figures 6–4 and 6–5, and in eqn (6.7). Also it may be convenient to refer to a diagram in Figure 7–2. We will generalize the results to a multilayer case in sec. 7.5. We start with the gradient of the pattern error as in eqn (3.4), however, we will calculate the gradient layer by layer, starting from the last/output layer.

### 7.2.1 The Last (output) Layer

The output layer, for each datapoint, calculates:

$$\hat{\mathbf{x}}_n^o = \begin{bmatrix} \mathbf{z}_n^h \\ 1 \end{bmatrix}; \quad \mathbf{u}_n = W^o \cdot \hat{\mathbf{x}}_n^o; \quad \mathbf{z}_n = \sigma(\mathbf{u}_n); \quad \mathbf{e}_n = \mathbf{z}_n - \mathbf{y}_n \quad (7.7)$$

The dimensionality of the output vector from the hidden layer,  $\mathbf{z}_n^h$ , is  $D^h$ , while vectors  $\mathbf{u}_n, \mathbf{z}_n, \mathbf{e}_n, \mathbf{y}_n$  are  $M$ -dimensional. Note that  $\sigma(\mathbf{u})$  is a vector in which the non-linear function  $\sigma(\cdot)$  is applied to each component of  $\mathbf{u}$ . A collection of possible non-linear functions, aka activation functions, was discussed in sec. 6.4. In order to calculate the pattern gradient wrt to components of the output weight matrix  $W^o$ , we vectorize the matrix as:

$$\mathbf{w}^o = \text{vec}(W^o) = \downarrow W^o \quad (7.8)$$

The dimensionality of the vector  $\mathbf{w}^o$  is equal to  $M \times (D^h + 1)$ . Note that the input vector to the output layer,  $\mathbf{x}^o$ , is independent of  $\mathbf{w}^o$  and, following eqn (3.4), we can calculate the pattern gradient as:

$$\nabla E_n(\mathbf{w}^o) = \frac{\partial \mathbf{e}_n^T}{\partial \mathbf{w}^o} \cdot \mathbf{e}_n = \frac{\partial \mathbf{z}_n^T}{\partial \mathbf{w}^o} \cdot \mathbf{e}_n = \mathcal{J}_n(\mathbf{z}_n, \mathbf{w}^o) \cdot \mathbf{e}_n \quad (7.9)$$

The subscript  $n$  refers to the  $n$ th data point, the superscript  $o$  refers to the parameters of  $W^o$  in the output layer. Using the chain rule of differentiation, and eqn (7.7), the Jacobian can be further expressed as:

$$\mathcal{J}_n(\mathbf{z}_n, \mathbf{w}^o) = \frac{\partial \mathbf{z}_n^T}{\partial \mathbf{w}^o} = \frac{\partial \mathbf{u}_n^T}{\partial \mathbf{w}^o} \cdot \text{diag}\left(\frac{d\sigma}{d\mathbf{u}_n}\right) = \mathcal{J}_n(\mathbf{u}_n, \mathbf{w}^o) \cdot \text{diag}(\boldsymbol{\sigma}'_n) \quad (7.10)$$

where

$$\boldsymbol{\sigma}'_n = \frac{d\sigma}{d\mathbf{u}_n}$$

is a  $D^h$ -component vector of derivatives of the activation function wrt to its input vector  $\mathbf{u}$ . The methods of calculating such derivatives were discussed in sec. 6.4. For example, if

$$z = \sigma(u) = \text{atanh}(u) \quad , \quad \text{then} \quad \frac{d\sigma}{du} = \sigma' = 1 - z^2$$

Combining the above results together we can re-write eqn (7.9) for the pattern gradient in the following form:

$$\nabla E_n(\mathbf{w}^o) = \mathcal{J}_n(\mathbf{u}_n, \mathbf{w}^o) \cdot \text{diag}(\boldsymbol{\sigma}'_n) \cdot \mathbf{e}_n = \mathcal{J}_n(\mathbf{u}_n, \mathbf{w}^o) \cdot \boldsymbol{\delta}_n \quad (7.11)$$

where

$$\boldsymbol{\delta}_n = \text{diag}(\boldsymbol{\sigma}'_n) \cdot \mathbf{e}_n = \boldsymbol{\sigma}'_n \odot \mathbf{e}_n \quad (7.12)$$

are called the **delta errors**.

Note that if we have two vectors of the same size, say  $\mathbf{a}, \mathbf{b}$  then

$$\text{diag}(\mathbf{a}) \cdot \mathbf{b} = \mathbf{a} \odot \mathbf{b}$$

where  $\odot$  denotes the component-wise multiplication of two vectors.

Following considerations of sec. 2.2 we note that the Jacobian in eqn (7.11) is independent of  $\mathbf{w}^o$ , and since  $\mathbf{u} = W^o \cdot \mathbf{z}^h$  and using eqn (2.17) we have

$$\nabla E_n(\mathbf{w}^o) = \mathcal{J}_n(\mathbf{u}_n, \mathbf{w}^o) \cdot \boldsymbol{\delta}_n = \downarrow(\boldsymbol{\delta}_n \cdot (\hat{\mathbf{x}}_n^o)^T) \quad (7.13)$$

Re-shaping the pattern gradient into the equivalent matrix form and following eqn (3.5) we finally have the expression for the upgrade of the output weight matrix  $W^o$

$$\Delta W_n^o = -\eta \cdot \boldsymbol{\delta}_n \cdot (\hat{\mathbf{x}}_n^o)^T \quad (7.14)$$

It is an important result stating that the weight update in the pattern mode is proportional to the **outer product** of the **delta errors** and the input vector  $\hat{\mathbf{z}}_n^h$ , with  $\eta$  being the learning rate parameter.

It is perhaps a good place to say that one of the typical problem encountered during learning stems from the fact that the delta errors are products of the errors and derivatives of the non-linear function  $\sigma$ . If the derivatives are zero, e.g. for the large values of the input signal, no learning takes place since  $\Delta W_n^o = 0$ , see eqn (7.14).

### 7.2.2 The Hidden Layer

The hidden layer, for each datapoint, calculates:

$$\hat{\mathbf{x}}_n = \begin{bmatrix} \mathbf{x}_n \\ 1 \end{bmatrix}; \quad \mathbf{u}_n^h = W^h \cdot \hat{\mathbf{x}}_n; \quad \mathbf{z}_n^h = \varphi(\mathbf{u}_n^h) \quad (7.15)$$

The dimensionality of the vector  $\mathbf{x}_n$  is  $D$ , while vectors  $\mathbf{u}_n^h, \mathbf{z}_n^h$  are  $D^h$ -dimensional. As before we vectorize the hidden weight matrix  $W^h$ :

$$\mathbf{w}^h = \text{vec}(W^h) = \downarrow W^h \quad (7.16)$$

The dimensionality of the vector  $\mathbf{w}^h$  is equal to  $D^h \times (D + 1)$ . The pattern gradient wrt  $\mathbf{w}^h$  can be calculated as in eqn (7.9)

Calculation of the gradient follows eqn (7.9) with  $\mathbf{w}^h$  replacing  $\mathbf{w}^o$

$$\nabla E_n(\mathbf{w}^h) = \frac{\partial \mathbf{e}_n^T}{\partial \mathbf{w}^h} \cdot \mathbf{e}_n = \frac{\partial \mathbf{z}_n^T}{\partial \mathbf{w}^h} \cdot \mathbf{e}_n = \frac{\partial \mathbf{u}_n^T}{\partial \mathbf{w}^h} \cdot \text{diag}(\boldsymbol{\sigma}') \cdot \mathbf{e}_n = \frac{\partial \mathbf{u}_n^T}{\partial \mathbf{w}^h} \cdot \boldsymbol{\delta}_n = \mathcal{J}_n(\mathbf{u}_n^h, \mathbf{w}^h) \cdot \boldsymbol{\delta}_n \quad (7.17)$$

where the  $n$ -th delta error is specified in eqn (7.12). Note that initially the gradient calculations used the output signals  $\mathbf{z}_n$  and  $\mathbf{e}_n$  that, by using the chain rule of differentiation, has been replaced by  $\mathbf{u}_n$  and  $\boldsymbol{\delta}_n$  which are closer to the hidden layer.

To continue calculations of the gradient as in eqn (7.17) we first isolate the non-bias component in  $\mathbf{u}_n$  following the notation used in eqn (6.5)

$$\mathbf{u}_n = W^o \cdot \hat{\mathbf{z}}_n^h = \bar{W}^o \cdot \mathbf{z}_n^h + \mathbf{b}^o$$

After transposition we have

$$\mathbf{u}_n^T = \mathbf{z}_n^{hT} \cdot \bar{W}^{oT} + \mathbf{b}^{oT}$$

Now, knowing that  $\mathbf{b}^o$  is independent of  $\mathbf{w}^h$ , we can continue with the gradient calculations as follows:

$$\nabla E_n(\mathbf{w}^h) = \frac{\partial \mathbf{u}_n^T}{\partial \mathbf{w}^h} \cdot \boldsymbol{\delta}_n = \frac{\partial \mathbf{z}_n^{hT}}{\partial \mathbf{w}^h} \cdot \bar{W}^{oT} \cdot \boldsymbol{\delta}_n = \frac{\partial \hat{\mathbf{z}}_n^{hT}}{\partial \mathbf{w}^h} \cdot \mathbf{e}_n^h = \mathcal{J}_n(\mathbf{z}_n^h, \mathbf{w}^h) \cdot \mathbf{e}_n^h \quad (7.18)$$

where

$$\mathbf{e}_n^h = \bar{W}^{oT} \cdot \boldsymbol{\delta}_n = \bar{W}^{oT} \cdot (\boldsymbol{\sigma}' \odot \mathbf{e}_n) \quad (7.19)$$

is the equivalent error as seemed at the hidden layer, that is, the output **error**  $\mathbf{e}_n$  **back-propagated** through the output layer described by  $\bar{W}^o$  and  $\sigma$ .

Eqn (7.18) is exactly equivalent to eqn (7.9), hence, following considerations for the output layer we can define:

The derivative of the nonlinear function in the hidden layer:

$$\boldsymbol{\varphi}'_n = \frac{d\varphi}{d\mathbf{u}_n^h}$$

The delta error in the hidden layer:

$$\boldsymbol{\delta}_n^h = \text{diag}(\boldsymbol{\varphi}'_n) \cdot \mathbf{e}_n^h = \boldsymbol{\varphi}'_n \odot \mathbf{e}_n^h \quad (7.20)$$

The pattern gradient wrt  $\mathbf{w}^h$

$$\nabla E_n(\mathbf{w}^h) = \mathcal{J}_n(\mathbf{u}_n^h, \mathbf{w}^h) \cdot \boldsymbol{\delta}_n^h = \downarrow(\boldsymbol{\delta}_n^h \cdot \hat{\mathbf{x}}_n^T) \quad (7.21)$$

and the update of the weight matrix in the hidden layer

$$\Delta W_n^h = -\eta \cdot \boldsymbol{\delta}_n^h \cdot \hat{\mathbf{x}}_n^T \quad (7.22)$$

### 7.2.3 Summary of learning in two-layer nnet

The block diagram of a two-layer nnet showing its feedforward part and the learning, that is, feedback part is shown in Fig. 7-2.

**For each input pattern**, the feedforward part calculates the signals as outlined in the previous sections, namely

$$\mathbf{x}_n \rightarrow \mathbf{u}_n^h \rightarrow \mathbf{z}_n^h = \mathbf{x}_n^o \rightarrow \mathbf{u}_n \rightarrow \mathbf{z}_n$$

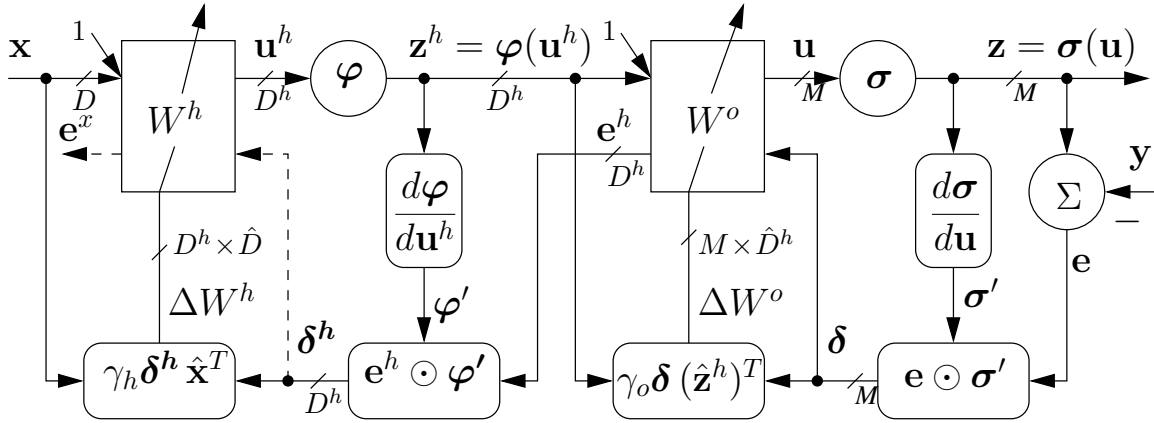


Figure 7-2: The feedforward and feedback parts of a two-layer nnet

In the learning path we calculate the vectors derivatives of the nonlinear functions:

$$\sigma'_n \text{ and } \varphi'_n$$

and then the error vectors, namely, delta errors and back-propagated error

$$(\mathbf{e}_n^x \leftarrow) \delta_n^h \leftarrow \mathbf{e}_n^h \leftarrow \delta_n \leftarrow \mathbf{e}_n$$

Note that according to eqn (7.19) the delta errors are back-propagated through the relevant weight matrices. The  $\mathbf{e}_n^x$  error is not used in calculations but is shown to illustrate the concept of back-propagation.

Finally, two weight updates are calculated

$$W^o \text{ and } W^h$$

## 7.3 Pattern and batch learning

In the previous sections we concentrated on the pattern learning. The conceptual objective of the learning procedure is to arrive at the values of network parameter, the weight matrices in this case, such that the total error aka the performance or loss function of eqns (7.2), (7.3) is minimised to the satisfactory level. This is going to be achieved by updating the weight



matrices as described in eqns (7.14) and (7.22) for all input patterns available. We refer to one pass through all pattern as one **learning epoch**. Typically a number of epochs is required to reduce the error to the desired level. We will highlight this problem in examples.

Eqns (7.14) and (7.22) describe the pattern learning in which we aim at minimizing the pattern error of eqn (7.2) that should eventually result in minimisation of the total error as in eqn (7.3). In the **batch learning**, the pattern weight updates are accumulated for the whole epoch and performed once per epoch.

$$\Delta W^o = \sum_{n=1}^N \Delta W_n^o = -\eta \sum_{n=1}^N \delta_n \cdot (\hat{\mathbf{x}}_n^o)^T = -\eta_z S^o \cdot \hat{X}^{oT} \quad (7.23)$$

where  $S^o$  and  $X^o$  are matrices collecting all delta errors and outputs from the hidden layer, respectively, that is:

$$S^o = [\delta_1 \dots \delta_N] , \quad \hat{X}^o = [\hat{\mathbf{x}}_1^o \dots \hat{\mathbf{x}}_N^o]$$

Similarly for the hidden layer we calculate:

$$\Delta W^h = \sum_{n=1}^N \Delta W_n^h = -\eta \sum_{n=1}^N \delta_n \cdot (\hat{\mathbf{x}}_n)^T = -\eta_h S^h \cdot \hat{X}^T \quad (7.24)$$

where  $S^h$  is the matrix collecting all hidden delta errors:

$$S^h = [\delta_1^h \dots \delta_N^h]$$

Some points to consider:

**Weight Initialisation** The weight are initialised in one of the following ways:

- using prior information if available. The Nguyen-Widrow algorithm presented used in MATLAB function **initnw** is a good example of such initialisation.
- to **small** uniformly distributed random numbers.

Incorrectly initialised weights cause that the activation potentials may become large which saturates the neurons. In saturation, derivatives  $\sigma' = 0$  and no learning takes place.

A good initialisation can significantly speed up the learning process.

**Randomisation** For the pattern learning it might be a good practice to randomise the order of presentation of learning examples between epochs.

**Validation** In order to validate the process of learning the available data is randomly partitioned into a **learning set** which is used for learning, and a **test set** which is used for validation of the obtained data model.

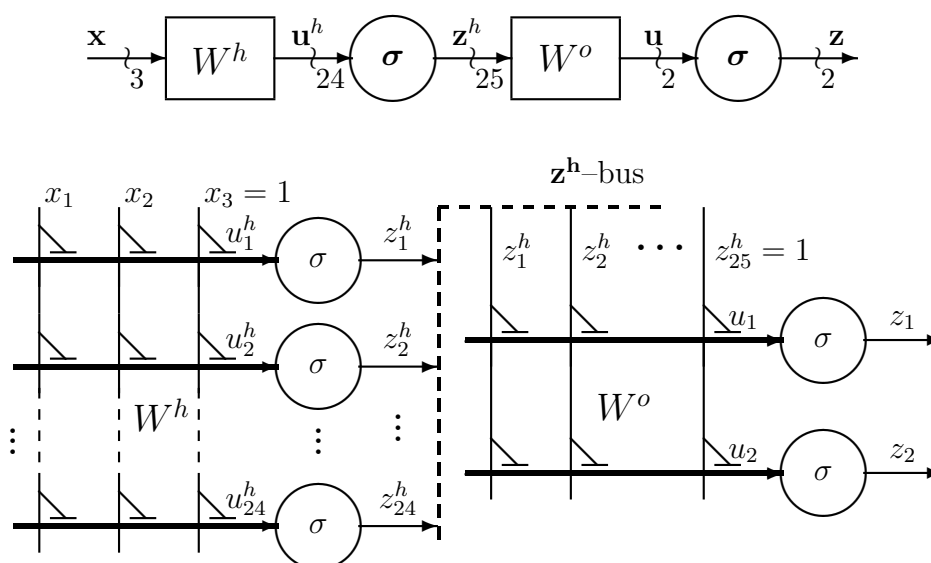
## 7.4 (Simple) example of function approximation

(The full MATLAB code is in <http://users.monash.edu/~app/Lrn/fap2D.m> )

We consider a two-layer nnet as in Fig. 7-2 that is going to approximate two functions of two variables. The network parameters are as follows:

```
D = 2 ; % Dimensionality of input signals
Dh = 24; % Dimensionality of hidden signals
M = 2 ; % Dimensionality of outputs
```

The relevant structure can be visualised in the following way:



As non-linear functions we selected the *sigmoidal* and *atan* functions. See sec. 6.4 for details. Two functions to be approximated by the two-layer nnet are as follows:

$$y_1 = x_1 e^{-\rho^2}, \quad y_2 = \frac{\sin 2\rho^2}{4\rho^2}, \quad \text{where } \rho^2 = x_1^2 + x_2^2$$

The domain of the function is a square  $x_1, x_2 \in [-2, 2)$ .

In order to form the learning set the functions are sampled on a regular  $16 \times 16$  grid so that the number of datapoints is  $N = 256$ . Subsequently for each data point  $\mathbf{x}_n = [x_{1n}, x_{2n}]^T$  we calculate  $\mathbf{y}_n = [y_{1n}, y_{2n}]^T$  using the expressions above. Finally we can form matrices  $\hat{X}$  and  $Y$  that look as follows

```
Xh = -2.00   -2.00   ...   1.75   1.75
      -2.00   -1.75   ...   1.50   1.75
      1.00    1.00   ...   1.00   1.00

Y = -0.0007  -0.0017   ...   0.0086   0.0038
     -0.0090   0.0354   ...  -0.0439  -0.0127
```

The functions to be approximated are plotted side-by-side in Fig. 7-3. Note that in reality the functions are defined over the same domain, namely,  $x_1, x_2 \in [-2, 2)$ , hence are superimposed on each other.

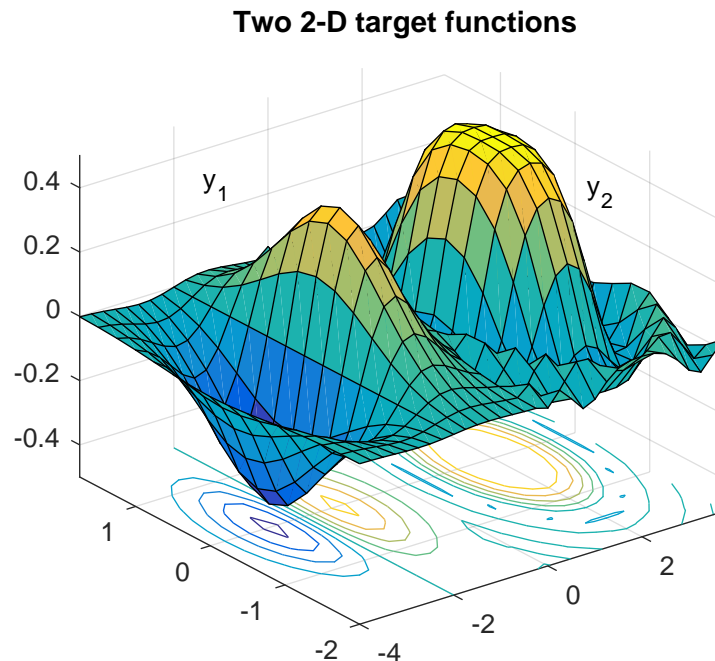


Figure 7-3: Two functions to be approximated

Having the datapoints representing functions to be approximated, and the structure of the nnet, we can perform the learning steps. In the example we use the batch learning as described in sec. 7.3 and the relevant code is as follows:

```

for ep = 1:nepchs % the epoch loop
% The forward pass
  Uh = Wh*Xh; % hidden post synaptic
  % Hidden signals (Dh by N)
  Zh = ones(Dh, N)/(1+exp(-Uh)); % sigmoidal nonlinearity
  dZh = Zh.*(1-Zh); % Derivatives of hidden signals
  Zhh = [Zh ; ones(1,N)] ; % appending the bias input
  Z = tanh(Wz*Zhh) ; % Output signals (M by N)
  dZ = 1 - Z.^2 ; % Derivatives of output signals
% The backward pass
  Ey = Z - Y; % The output errors (M by N)
  % the performance function: total mse
  Er(ep) = sum(sum(Ey.^2)/2)/N ; % mse after each epoch
  delY = Ey.*dZ; % Output delta signal (M by N)

```

```

dWz = -delY*Zhh';           % Update of the output matrix M by Dh+1
Eh = Wz(:,1:end-1)']*delY; % The backpropagated hidden errors (Dh by N)
delH = Eh.*dZh ;           % Hidden delta signals (Dh by N)
dWh = -delH*Xh';           % Update of the hidden matrix
% The batch update of the weights:
Wz = Wz + gamma(1)*dWz ; Wh = Wh + gamma(2)*dWh ;
end % of the epoch loop

```

You should compare and match the lines of code with expressions from sec. 7.3. The result of approximation is presented in Fig. 7-4.

**epoch: 2000, error: 0.0043, gamma: 0.0010 0.0350**

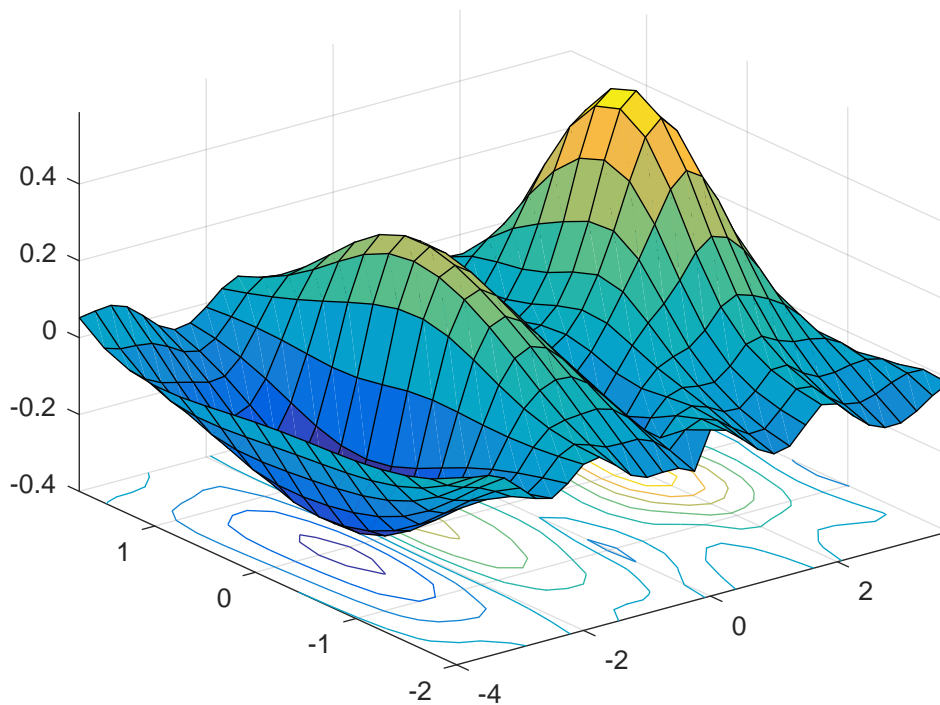


Figure 7-4: Two approximated functions

The convergence of the algorithm is monitored by the total mean squared error that the learning algorithm minimises. Evolution of the total mean squared error over learning epochs is presented in Fig. 7-5.

From the example we can see that there are a number of parameters to consider — the main are:

- The structure. The number of hidden layers and how many neurons in each layer. We have arbitrarily decided one hidden layer with 24 neurons.

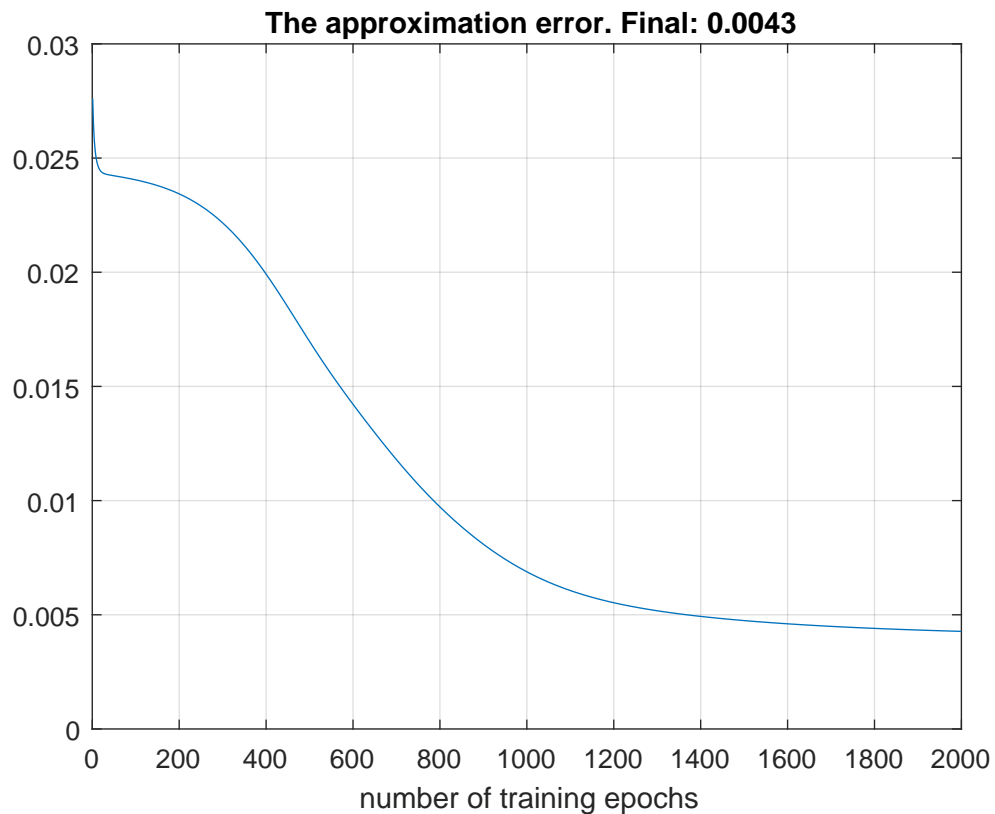


Figure 7–5: Evolution of the total mean squared error over learning epochs

- The nonlinear functions. We have arbitrarily selected the sigmoidal and the hyperbolic tangent functions.
- The learning algorithm. We selected the gradient decent algorithm. Other options are considered in sec. 3. In sec. 8 we consider a stochastic gradient algorithm which is popular in the classification networks. In any case the concept of **error backpropagation** plays the fundamental role.

## 7.5 Learning in deep neural networks

In this section we will generalise algorithms developed for two-layer neural network into the multilayer case recently referred to as deep neural networks. The deep neural network will consist of  $K$  layers. We start with repeating eqn (6.6) describing the deep architecture.

In the **feedforward pass**, we calculate, layer by layer, postsynaptic activities,  $\mathbf{u}^{(k)}$ , and the output signals from each layer,  $\mathbf{z}^{(k)}$ :

$$\mathbf{x}_n^{(k)} = \mathbf{z}_n^{(k-1)} \quad , \quad \hat{\mathbf{x}}_n^{(k)} = \begin{bmatrix} \mathbf{x}_n^{(k)} \\ 1 \end{bmatrix} \quad , \quad \mathbf{u}_n^{(k)} = W^{(k)} \cdot \hat{\mathbf{x}}_n^{(k)} \quad , \quad \mathbf{z}_n^{(k)} = \boldsymbol{\sigma}(\mathbf{u}_n^{(k)}) \quad (7.25)$$

In the **feedback pass**, we calculate the **delta errors** as in eqns (7.12) and (7.20):

$$\boldsymbol{\delta}_n^{(k)} = \boldsymbol{\sigma}'^{(k)} \odot \mathbf{e}_n^{(k)} \quad , \quad \text{for } k = K, \dots, 1 \quad , \quad \mathbf{e}_n^{(K)} = \mathbf{e}_n = \mathbf{z}_n - \mathbf{y}_n \quad (7.26)$$

together with the **backpropagated errors** as in eqn (7.19)

$$\mathbf{e}_n^{(k-1)} = \bar{W}^{(k)T} \cdot \boldsymbol{\delta}_n^{(k)} \quad , \quad \text{for } k = K, \dots, 2 \quad (7.27)$$

Note that backpropagating error through the weight matrix as in eqn (7.27), we only take the non-bias part of the matrix,  $\bar{W}^{(k)}$ .

For each layer, we can now calculate the weight updates as discussed in the previous sections, either in the **pattern mode**:

$$\Delta W_n^{(k)} = -\eta \cdot \boldsymbol{\delta}_n^{(k)} \cdot (\hat{\mathbf{x}}_n^k)^T \quad (7.28)$$

or in the **batch mode**

$$\Delta W^{(k)} = -\eta_z S^{(k)} \cdot \hat{X}^{(k)T} \quad (7.29)$$

where  $S^{(k)}$  and  $X^{(k)}$  are matrices collecting delta errors and inputs to the given layer for all patterns, respectively, that is:

$$S^{(k)} = [\boldsymbol{\delta}_1^{(k)} \quad \dots \quad \boldsymbol{\delta}_N^{(k)}] \quad , \quad \hat{X}^{(k)} = [\hat{\mathbf{x}}_1^{(k)} \quad \dots \quad \hat{\mathbf{x}}_N^{(k)}]$$

## 8 Softmax Classifier

In sec. 1.3 we briefly mentioned classification aka pattern/object recognition as an example of supervised clustering.

In the spirit of “deep learning” the network for object classification, aka **classifier**, consists of  $K$  neuronal layers as discussed in the previous sections, in particular in 6.6, 7.1 and 7.5.

The final layer of the classifier has the number of outputs  $\mathbf{z}$  equal to the number of expected classes, say  $M$ . Traditionally, the objects to be classified are represented by feature vectors  $\mathbf{x}_n$ . For each such vector, the expected output vector  $\mathbf{y}_n$  has unity at the position, say  $m$ , equivalent to the expected class label, and zero at all other positions, e.g.

$$\mathbf{y}_n = [0 \ 0 \ 1 \ 0 \ 0 \ 0]^T, \quad M = 6, \quad m = 3$$

The real output would have maximum value at that position:

$$m = \arg \max(\mathbf{z}_n)$$

$m = 3$  in the example.

It seems more natural, to organize the real output to show the probabilities  $\mathbf{z}_n$  that the output  $z_{nm}$  recognizes the input pattern with the label  $m$ . We could start with using sigmoidal logistic function at all outputs (see sec. 6.4) so that the output values would be between zero and one. Typically, to have a probability-like values we use a **softmax function** which is a generalization of the logistic function. The softmax function is defined in the following way

$$\mathbf{z}_n = S(\mathbf{u}_n) = \frac{e^{\mathbf{u}_n}}{\mathbf{1}^T \cdot e^{\mathbf{u}_n}}, \quad \text{or} \quad z_{nm} = \frac{e^{u_{nm}}}{\sum_{i=1}^M e^{u_{ni}}}, \quad \text{for } m = 1, \dots, M \quad (8.1)$$

where  $\mathbf{u}_n = W^o \cdot \hat{\mathbf{z}}_n^{(K-1)}$ , is a postsynaptic activity at the **final layer** of the neural network for the  $n$ th pattern, see sec. 7.5, and  $\mathbf{1}$  is a vector of all ones. It is easy to note that

$$\sum_{m=1}^M z_{nm} = 1$$

In other words, the output of the softmax function  $\mathbf{z}_n$  for each pattern  $n$ , represents a **probability distribution** over  $M$  different possible outcomes of the classification process.

The **Softmax Classifier** uses the **cross-entropy**, that is, a negative logarithm of probabilities, as its loss function. The cross entropy is “gated” by the expected probabilities  $\mathbf{y}_n$ . Hence we have:

$$L_n = -\log \frac{e^{\mathbf{u}_n \odot \mathbf{y}_n}}{\mathbf{1}^T \cdot e^{\mathbf{u}_n}} = \log(\mathbf{1}^T \cdot e^{\mathbf{u}_n}) - \mathbf{u}_n \odot \mathbf{y}_n \quad (8.2)$$

where  $\mathbf{u}_n \odot \mathbf{y}_n$  is a component-wise multiplication of two vectors. Note that  $\mathbf{y}_n$  is a vector having a unity on position  $m$ . Therefore,  $\mathbf{u}_n \odot \mathbf{y}_n$  has only one non-zero component at the position  $m$ .

In order to perform the **gradient decent** we calculate first the derivative of the loss function with respect to the postsynaptic activities  $\mathbf{u}_n$ . Differentiation of eqn (8.2) yields:

$$\frac{\partial L_n}{\partial \mathbf{u}_n} = \frac{e^{\mathbf{u}_n}}{\mathbf{1}^T \cdot e^{\mathbf{u}_n}} - \mathbf{y}_n = \mathbf{z}_n - \mathbf{y}_n = \mathbf{e}_n \quad (8.3)$$

This is a very elegant results that states that the derivatives of the loss function are equal to probabilities  $z_{ni}$  for the  $i \neq m$  and  $z_{ni} - 1$  for the output matching the label of the pattern  $\mathbf{x}_n$ . In other words, the derivative of the loss function wrt to postsynaptic activity as in eqn (8.3), is simply equal to the  $n$ th error as specified in eqn (1.4).

Finally, we can calculate the gradient of the loss function wrt to weights. Let the final layer performs the operations as described above:

$$\mathbf{u}_n = W^o \cdot \hat{\mathbf{x}}_n^o, \quad \mathbf{z}_n = S(\mathbf{u}_n) \quad (8.4)$$

where, for simplicity,  $\mathbf{x}_n^o = \mathbf{z}_n^{(K-1)}$  is the input to the final layer. The gradient is:

$$\frac{\partial L_n}{\partial W^o} = \frac{\partial L_n}{\partial \mathbf{u}_n} \cdot \frac{\partial \mathbf{u}_n}{\partial W^o} = \mathbf{e}_n \cdot \mathbf{x}_n^{oT} \quad (8.5)$$

where the pattern error  $\mathbf{e}_n = \mathbf{z}_n - \mathbf{y}_n$ . Note the outer product of the error and input vectors. The pattern weight update is now

$$\Delta W_n^o = -\eta_z \cdot \mathbf{e}_n \cdot \mathbf{x}_n^{oT} \quad (8.6)$$

This equation is equivalent to eqn (7.28) for the **pattern learning**. Similarly for the **batch mode** we can adopt the eqn (7.29) in the following form:

$$\Delta W^o = -\eta_z S^o \cdot \hat{X}^{oT} \quad (8.7)$$

where  $S^o$  and  $\hat{X}^o$  are matrices collecting delta errors and inputs to the given layer for all patterns, respectively, that is:

$$S^o = [\mathbf{e}_1 \dots \mathbf{e}_N], \quad \hat{X}^o = [\hat{\mathbf{x}}_1^o \dots \hat{\mathbf{x}}_N^o]$$

It has been shown that the convergence of the learning algorithms with the softmax function works better if we add to the loss function of eqn (8.2) the **regularization term** equal to:

$$L_R = \frac{1}{2} \mathbf{w}^{oT} \cdot \mathbf{w}^o \quad (8.8)$$

where

$$\mathbf{w}^o = \text{vec}(W^o) = \downarrow W^o$$

It is easy to notice that the gradient of the regularization term,  $L_R$  is simply:

$$\frac{\partial L_R}{\partial W^o} = W^o \quad (8.9)$$



We can modify the batch weight update eqn (8.7) to become:

$$\Delta W^o = -\eta_z S^o \cdot \hat{X}^{oT} - \lambda W^o \quad (8.10)$$

Similar modification can be done to the pattern learning eqn (8.6).

## 8.1 A linear Softmax classifier example

In this section we consider the simplest case of a linear softmax classifier. For each input pattern  $\mathbf{x}_n$ , the network calculates, for each  $m$ th output, probabilities  $\mathbf{z}_n$  that  $\mathbf{x}_n$  belongs to the class  $k$ . It is expected that for correct classification,  $z_{nm} \approx 1$  for the  $m$ th output and  $\approx 0$  for all other outputs. The working of the network is described as follows:

$$\mathbf{u}_n = W \cdot \hat{\mathbf{x}}_n, \quad \mathbf{z}_n = S(\mathbf{u}_n) \quad (8.11)$$

where  $S()$  is the softmax function as in eqn (8.1) and  $W$  is the weight matrix obtained during the learning procedure

It needs to be said that the linear classifier can work well only for the **linearly separable** patterns. We explain this concept below.

For  $N$  patterns of dimensionality  $D$  belonging to  $M$  classes, we can re-write eqn (8.11)

$$U = W \cdot \hat{X}, \quad Z = S(U)$$

where  $\hat{X}$  is  $(D+1) \times N$ ,  $W$  is  $M \times (D+1)$  and  $U, Z$  are  $M \times N$ .

In the example the 2-D pattern consists of  $M = 5$  clusters of points, 20 points per cluster, so that  $N = 100$ , as in Fig. 8-1.

The learning part of the code is as follows:

```
% M neurons ; Weights initialization
% the weight matrix W will be M by D+1 (including the bias)
W = 0.1*(rand(M,D+1)-0.5) ;

% learning/learning meta parameters
reg = 1e-3 ; % lambda in regularization
eta = 5/N ; % learning rate
nepchs = 40 ; % number of learning epochs

L = zeros(nepchs,1) ; % total loss
LData = zeros(nepchs,1) ; % data loss

% learning
for epch=1:nepchs % the epoch loop
    % Calculating the scores (postsynaptic activities) M by N
    U = W*Xh ; % each column (stimulus) gives the class scores
                % corresponding to the M classes
    % the Softmax classifier -- the cross-entropy loss function
```

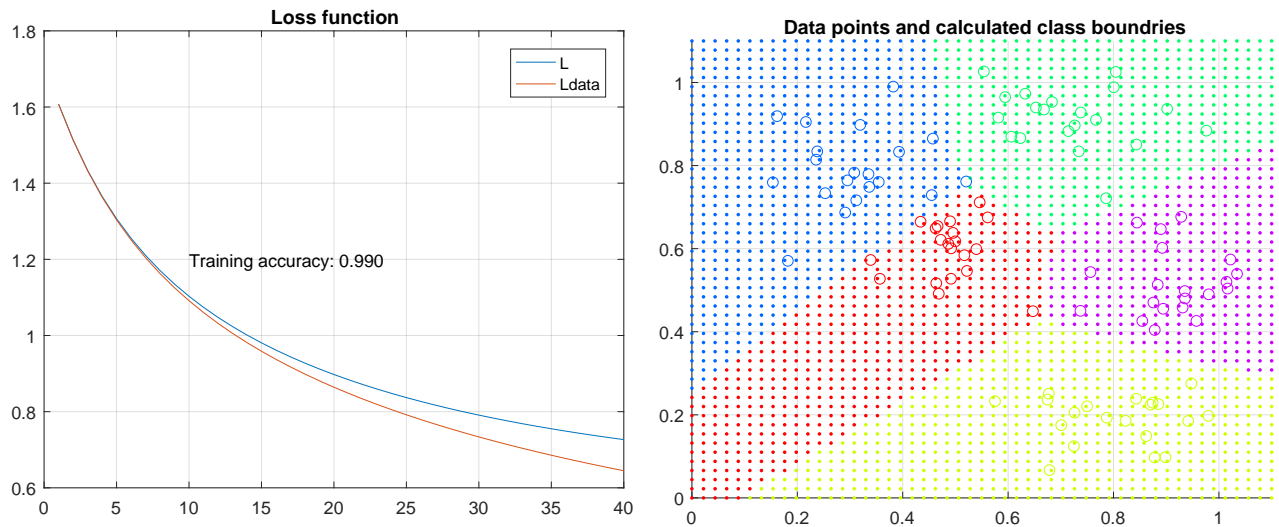


Figure 8–1: Left plot: evolution of the loss function  $L$  and the data loss function  $L_D$  over the learning epochs. Right plot: clusters of data points (circles) organized into  $M = 5$  linearly separable classes

```

ewx = exp(U) ;      % is M by N
s_ewx = sum(ewx) ; % is 1 by N
Pr = ewx./s_ewx ; % probabilities, M by N, outputs Z

% selecting the correct probability for each class m
pk = sum(Pr.*YY) ; % is 1 by N , class probabilities
% data loss is an average cross entropy
LData(epch) = -mean(log(pk)) ; % mean of -log(probs(k))
LReg = 0.5*reg*(W(:)'+W(:)) ; % regularization loss
L(epch) = LData(epch) + LReg ; % total loss
gL = Pr - YY ; % gradient of LD,
% The weight matrix update (including the regularization)
dW = gL*Xh' + reg*W ;
W = W - eta*dW ;
end % the epoch loop

% evaluate learning set accuracy
U = W*Xh ;
[~, Umx] = max(U) ;
tsacc = sum(Umx == Y)/N ;

```

The linear classifier works well only when the patterns are linearly separable. If this is not the case. The results might look like those in Fig. 8–2.

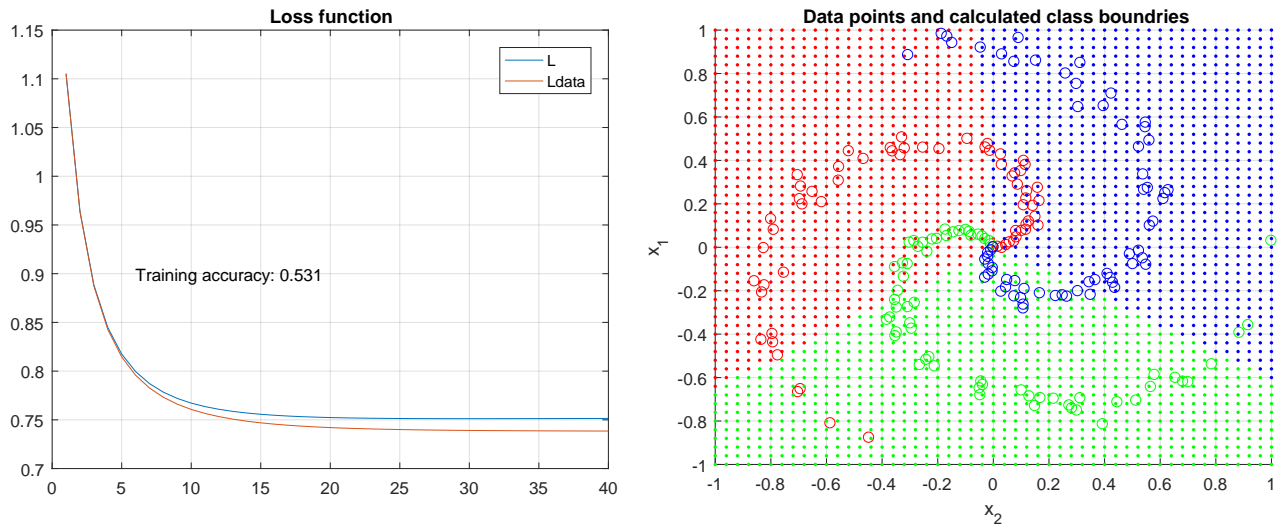


Figure 8–2: Left plot: evolution of the loss function  $L$  and the data loss function  $L_D$  over the learning epochs. Right plot: classification of noisy spiral data which is not linearly separable

The three spirals are clearly not linearly separable. From the left plot in Fig. 8–2 it can be seen that the loss function is nicely minimised, however, the learning accuracy is approximately 50%, that is, about half of the points are misclassified. The right plot in Fig. 8–2 shows the location of points in three areas of classification. Equations of the straight lines separating the regions are given by the weight matrix

$$W \cdot \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix} = 0$$

In our case the weight matrix scaled by the central column is:

$$W = \begin{bmatrix} -2.20 & 1 & 0.05 \\ 0.01 & 1 & -0.01 \\ 1.70 & 1 & -0.01 \end{bmatrix}$$

so the equations of three lines can be approximated as:

$$-2.2x_1 + x_2 = 0, \quad x_2 = 0, \quad 1.7x_1 + x_2 = 0$$

## 8.2 A two-layer Softmax classifier spiral data example

In order to deal with patterns that cannot be separated by straight lines, or hyperplanes in the case of higher dimensionality, we need to add nonlinear neuronal layers. For the example of a 3-spiral pattern, we use a two-layer network as described in sec. 7.2. In the classification multilayer networks it is typical to use the **ReLU nonlinearity** (see sec. 6.4) in hidden layer(s) and to softmax at the output layer.

There are two basic ways to perform the learning procedure, in the **pattern mode** or the **batch mode**.

In the case of the pattern learning mode, the weights are updated after each pattern is applied to the network as in eqn. (7.28). The complete code is given in <http://users.monash.edu/~app/Lrn/Mtlb/sftMx2LyrPt.m>

In the case of the batch learning mode, the weights are updated once per epoch, for all  $N$  patterns as in eqn. (7.29). The complete code is given in <http://users.monash.edu/~app/Lrn/Mtlb/sftMx2Lyr.m>

The learning part of the code is given below. Different lines of code for two modes are shown side-by-side. The same line of codes are not repeated for two modes.

```

                                % learning parameters
% pattern learning                % batch learning
etah = 5/N ;                      etah = 0.53/N ; % learning rate (hidden)
etaz = 1/N ;                      etaz = 0.09/N ; % learning rate (output)
reg = 1e-2/N ;                   reg = 1e-4 ;    % lambda in regularization
nepchs = 3000 ;                  nepchs = 8000 ; % number of learning epochs

                                L = zeros(nepchs,1) ; % total loss
                                LData = zeros(nepchs,1) ; % data loss
logpk = zeros (N,1) ; % pattern log(pk_n)
                                for epch=1:nepchs % the epoch loop
for n=1:N % pattern loop
    % evaluate all signals in the 2-layer Neural Network
    % first layer
    % postsynaptic activities in the 1st (hidden) layer
    uh = Wh*Xh(:,n) ;              Uh = Wh*Xh ;
    % ReLU nonlinearity and bias
    zh = [max(0,uh); 1];          Zh = [max(0,Uh); ones(1,N)];
    % second layer
    % Calculating the postsynaptic activities (scores)
    uz = Wz*zh ; % M by 1          Uz = Wz*Zh ; %M by Dh+1
    % the Softmax classifier -- the cross-entropy loss function
    ewx = exp(uz) ; % M by 1       ewx = exp(Uz) ; % M by N
    s_ewx = sum(ewx); % 1 by 1     s_ewx = sum(ewx); % is 1 by N
    % probabilities = outputs z_n
    Pr = ewx./s_ewx ; % M by 1     Pr = ewx./s_ewx ; % M by N
    % selecting the correct probability for each class -- class probabilities
    pk = sum(Pr.*YY(:,n)); % 1 by 1 pk = sum(Pr.*YY) ; % 1 by N

```

```

logpk(n) = log(pk) ;
           % gradient of LData
gL = Pr - YY(:,n) ;           gL = Pr - YY ;
           % The weight matrix update (including the regularization)
dWz = gL*zH' + reg*Wz ;       dWz = gL*Zh' + reg*Wz ;
           Wz = Wz - etaz*dWz ;
% backpropagation from 2nd layer to the 1st layer
delh = Wz(:,1:end-1)*gL ; % backprop error
           % backprop through ReLU
delh(uh<=0) = 0 ;             delh(Uh<=0) = 0 ;
% The weight matrix update (including the regularization)
dWh = delh*Xh(:,n)' + reg*Wh ; dWh = delh*Xh' + reg*Wh ;
           Wh = Wh - etah*dWh ;
end % pattern loop
           % data loss is an average cross entropy
           % mean of -log(probs(pk))
LData(epch) = -mean(logpk) ;   LData(epch) = -mean(log(pk)) ;
           % regularization loss
Lreg = 0.5*reg*(Wh(:)'*Wh(:) + Wz(:)'*Wz(:)) ;
L(epch) = LData(epch) + Lreg ; % total loss
end % the epoch loop

```

In the pattern mode the time taken was 3 msec/epoch, total  $\approx 9$  sec The results of learning are graphically presented in Fig. 8–3.

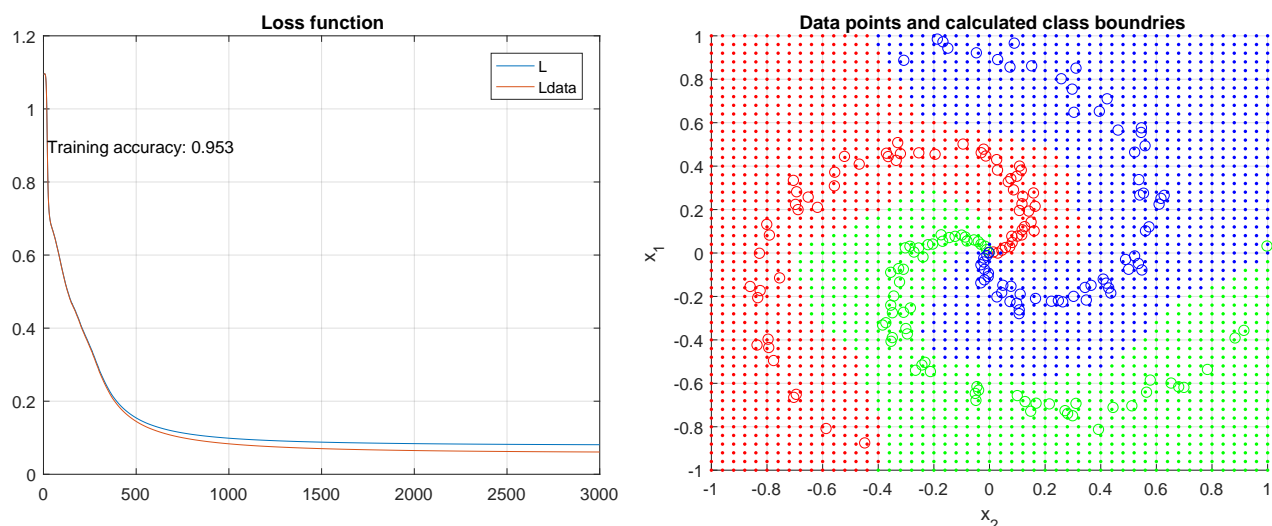


Figure 8–3: Left plot: evolution of the loss function  $L$  and the data loss function  $L_D$  over the learning epochs. Right plot: classification of noisy spiral data with two-layer neural network

In the case of the batch mode the time taken was 2 msec/epoch, total  $\approx 9$  sec. The results of learning are graphically presented in Fig. 8–4.

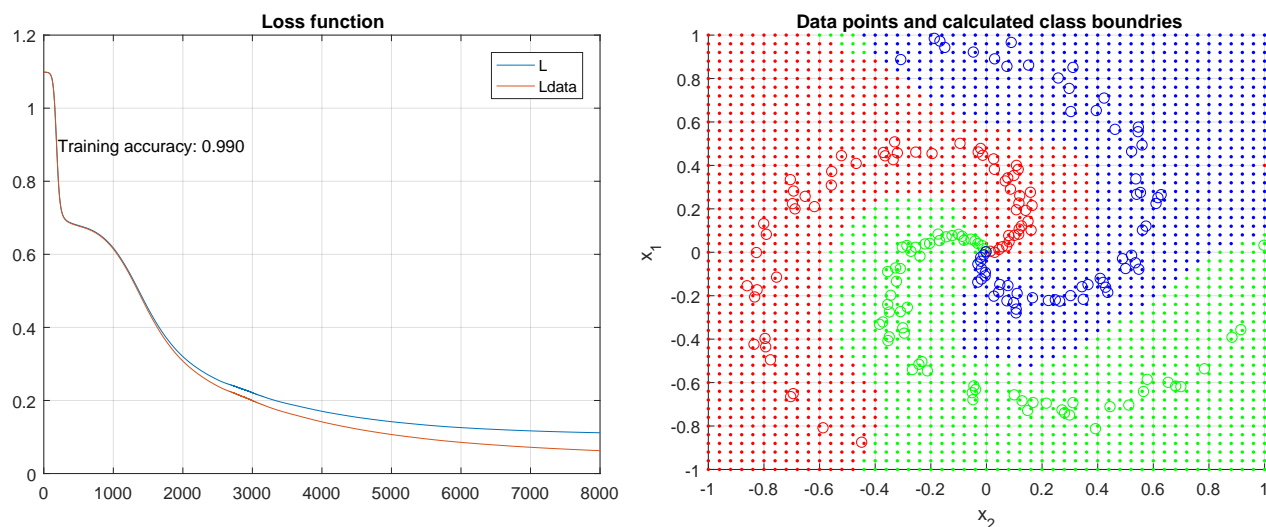


Figure 8–4: Left plot: evolution of the loss function  $L$  and the data loss function  $L_D$  over the learning epochs. Right plot: classification of noisy spiral data with two-layer neural network

Equations of three curves separating the patterns is given by:

$$F(x_1, x_2) = W^o \cdot \left[ \begin{array}{c} \max(0, W^h \cdot \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}) \\ 1 \end{array} \right] = 0 \quad (8.12)$$

where  $W^o$  is  $M \times (D^h + 1) = 3 \times 101$  and  $W^h$  is  $D^h \times (D + 1) = 100 \times 3$ , the total of 603 parameters.

The **learning accuracy** is **validated** by calculating the maximum class probabilities and comparing them with the expected class labels given by  $Y$ . The achieved learning accuracy for one particular learning session is 95.3%. It means that nine patterns have been miss-classified, since  $(N - 9)/N = 95.3\%$ .

In general, no fundamental differences between two modes of learning have been identified. In our example the pattern mode required less learning epochs, however the total learning time was similar.

It is a good opportunity to consider options that could increase the training accuracy. Some points to consider are as follows:

- For the two-layer case, we can increase the size of the hidden layer. More hidden neurons should result in the increased accuracy.
- We could “go deep” adding more hidden layer. It typically works.
- We can try to use more complex learning algorithm. We use the basic gradient descent algorithm. We could try, for example, the Adam algorithm described in sec. 4.5.

### 8.3 The MNIST example: Hand-written digits classification

We will use the same code as in the previous section to create a neural network that can recognize handwritten digits. The MNIST database of handwritten digits is available from <http://yann.lecun.com/exdb/mnist/>. It contains a training set of 60,000 examples, and a test set of 10,000 examples. Each example is a  $28 \times 28$  black-and-white image of a hand written characters.

The first problem to consider is how to represent an image in a form of an equivalent vector. We adopt the simplest example of stacking the columns of the image to create a vector of dimensionality  $D = 28 \times 28 = 784$ . We consider other possibilities later.

In our first experiment we train the network with only  $N = 5,000$  samples out of the total 60,000 to save on the computational time. As in the spiral case, we use a two-layer classifier with ReLU non-linearity in the hidden layer and the softmax at the output. The number of classes is equal to number of digits to be classified, that is,  $M = 10$ .

The number of hidden neurons has been set to  $D_h = 40$ . This time we use the **pattern mode**. The complete code can be found in <http://users.monash.edu/~app/Lrn/Mt1b/hndWrtnS.m>. We do not use the regularization term. During learning the loss function evolves as in Figure 8–5.

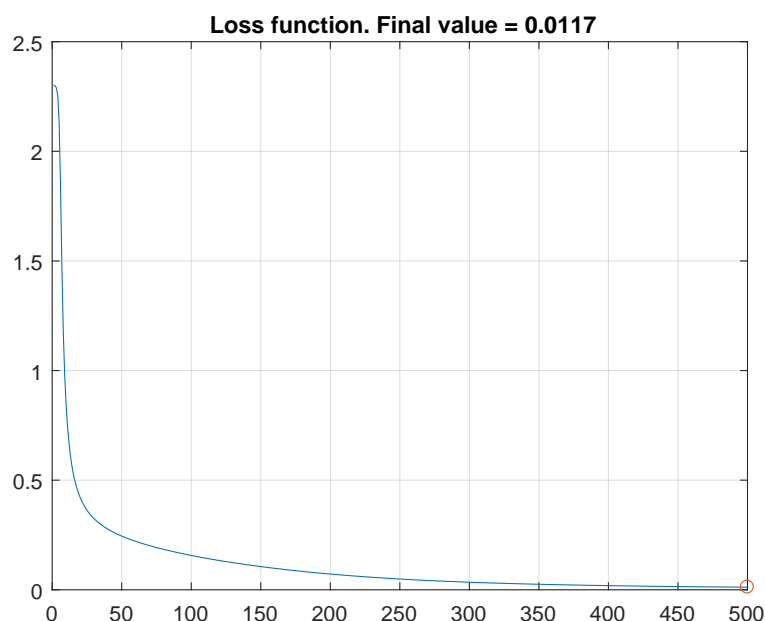


Figure 8–5: Evolution of the loss function  $L$  during training the network for the handwritten digit recognition

The training time for 500 epochs is just 239 sec on my Dell laptop. After 500 epochs we test the results for the training set of  $N = 5,000$  and the accuracy is 100%, that is, all the training patterns are correctly classified.

Now we test the network for all 60,000 training example and then for 10,000 testing examples. The code for validation can be found in <http://users.monash.edu/~app/Lrn/Mtlb/hndWrtnSVld.m>. The accuracy for both cases are 92.21% and 91.88%, respectively, or equivalently, the test error rates are 7.8% and 8.1%. It should be compared with the results reported in Y. LeCun, L. Bottou, Y. Bengio and P. Haffner: Gradient-Based Learning Applied to Document Recognition, Proceedings of the IEEE, 86(11):2278-2324, November 1998. The reported error rate was 4.5% for a network of  $D_h = 1000$  hidden neurons trained for all 60,000 examples and tested with 10,000 testing images.



## 9 Convolutional Neural Networks

### 9.1 Preliminary considerations

Convolutional neural networks (CNNs) are modifications of the deep neural networks considered in sec. 6.6 and 7.5. CNNs have become standard tools in cases when the input data consist of 2-D objects like images. In this case CNNs eliminate one of the fundamental problem, namely creation of the feature vectors that would represent objects in the image. It is expected that the convolutional layers will find out the best features automatically.

In the MNIST example of sec. 8.3 the feature vector was created in the simplest possible way just by stacking the columns of the image one upon the other. It is clearly visible that doing so we seem to be loosing the neighbourhood information about the pixels. The pixels that are adjacent in the image are located far-apart in the feature vector.

In this section we will present introductory concepts used in CNNs before formal definitions given in the next sections. We start with the concept of the **receptive field**. The neighbourhood of each pixel contributes to the neuronal output signal through a related set of weights organized into a **filter** aka **kernel** or **mask**. Typically, the filter is fixed for the given layer of the network and the output signal is created using a 2D convolution operation:

$$U = \text{conv2d}(X, W) = W \star X \quad (9.1)$$

where  $X$  represents an input image to the layer,  $U$  is the output image aka 2D feature map, and  $W$  is a 2D **filter** aka **mask** aka **kernel** aka **weight matrix**. The ‘ $\star$ ’ is often used as a convolution operator. The related MATLAB function is called **conv2**. The convolution is a linear operator and conceptually replaces operation  $\mathbf{u}_n^{(k)} = W^{(k)} \cdot \hat{\mathbf{x}}_n^{(k)}$  in eqn (7.25).

The simplest component of a convolutional layer might look like an example in Figure 9–1. It

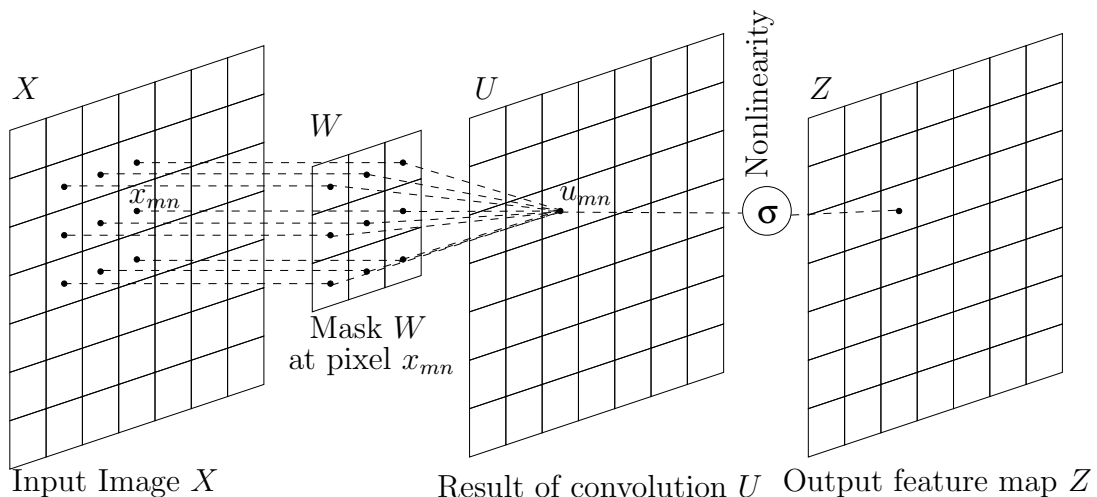


Figure 9–1: One layer of a simple convolutional neural network

is a 2D case, that is, a 2D 3×3 filter (kernel, mask, weight matrix)  $W$  is applied to an image  $X$  using convolution operation as in eqn (9.1). For each pixel  $x_{mn}$  and its eight neighbouring

pixels the  $m, n$  neuron calculates  $u_{mn}$  as a sum of products of pixel values and corresponding mask values:

$$\begin{aligned} u_{mn} &= x_{m-1,n-1} \cdot w_{-1,-1} + x_{m-1,n} \cdot w_{-1,0} + x_{m-1,n+1} \cdot w_{-1,1} \\ &+ x_{m,n-1} \cdot w_{0,-1} + x_{m,n} \cdot w_{0,0} + x_{m,n+1} \cdot w_{0,1} \\ &+ x_{m+1,n-1} \cdot w_{1,-1} + x_{m+1,n} \cdot w_{1,0} + x_{m+1,n+1} \cdot w_{1,1} \end{aligned} \quad (9.2)$$

or

$$U = W \star X$$

After the linear part of a neuronal layer  $U$ , a typical non-linearity  $\sigma$  is included, so that the output pixel (neuronal output) is calculated as:

$$z_{mn} = \sigma(u_{mn} + b), \text{ or } Z = \sigma(W \star X + B) \quad (9.3)$$

where  $\sigma$  is any suitable activation function described in sec. 6.4, ReLU being a recent popular choice,  $B$  being the bias for the layer.  $Z$  is typically referred to as a **feature map**.

The convolutional network of Figure 9–1 demonstrates two other concepts, typical to this type of networks, namely, **limited connectivity** and the **weight sharing**. Comparing Figure 9–1 and eqns (9.2), (9.3) with Figure 6–3 and eqns (6.3) we note that each neuron is calculating its output  $u_{mn}$  using input neurons from the **receptive field** only,  $3 \times 3 = 9$  in the example, rather than all input neurons as in the **full connectivity** case of the Figure 6–3. Additionally, each neuron in Figure 6–3 has its own weight vector, whereas in the CNN of 9–1 only one mask  $W^k$  is used for the layer, so that neurons share the weights.

The next complication stems from the fact that objects like colour images are represented by, typically, three 2D images. We can say that the colour image is represented by a 3D tensor. In general, at each layer of a convolutional network the input can be represented by  $P$ -dimensional tensor and the output can be a  $Q$ -dimensional tensor. Not to complicate the notation at this stage too much, we can write for each  $q$ -th output feature map

$$U_q = \text{convT}(X, W_q), \quad \text{for } q = 1 \dots Q \quad (9.4)$$

where  $X$  is a  $P$ -dimensional input tensor (e.g. colour image,  $P = 3$ ),  $W_q$  is a  $P$ -dimensional tensor filter, and **convT** represents a  $P$ -dimensional convolution operation. Related operation in MATLAB is called **convn**.

In general (and in practice) we can have a number of filters  $W^l$  in a given layer.

## 9.2 Convolution fundamentals

In this section we will cover details of two-dimensional convolution operation. You can skip this section if you are familiar with the contents.

In a 2D case, we have an image/feature map  $X$  of size  $R \times C$ : 
$$R \begin{array}{|c|} \hline X \\ \hline \end{array} \stackrel{C}{=} \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_R \end{bmatrix}$$

and a mask  $W$  of size  $r \times c$ :  $r \begin{array}{|c|} \hline c \\ \hline W \\ \hline \end{array}$ .

Each  $p, q$  pixel of the convolution of  $X$  and  $W$  can be calculated as:

$$U_{p,q} = (W \star X)_{p,q} = \sum_{m=0}^{r-1} \sum_{n=0}^{c-1} W_{m,n} X_{p-m,q-n} \quad (9.5)$$

The resulting size  $P \times Q$  of the convolution  $U$  depends on whether we allow the mask to be partially outside the image borders, assuming that the image is padded with enough zeros around its borders. If that is the case, then we have

$$p = 1, \dots, R + r - 1, \quad q = 1, \dots, C + c - 1, \quad P = R + r - 1, \quad Q = C + c - 1 \quad (9.6)$$

If, on the other hand, we require the mask to be entirely inside the image borders, then we have

$$p = r, \dots, R, \quad q = c, \dots, C, \quad P = R - r + 1, \quad Q = C - c + 1 \quad (9.7)$$

It maybe beneficial to know the difference between convolution and **cross-correlation**. Note that if we replace in eqn (9.5)  $m, n$  with  $-m, -n$ , namely

$$\mathbf{C}_{p,q} = \sum_{m=0}^{r-1} \sum_{n=0}^{c-1} W_{-m,-n} X_{p+m,q+n} \quad (9.8)$$

then the convolution becomes cross-correlation with the mask  $W$  being flipped over in both directions (left-right and up-down). Such a flipped mask can be calculated as follows:

$$V = J_r \cdot W \cdot J_c \quad (9.9)$$

where  $J$  is an anti-diagonal identity-like matrix.  $J$  matrix is a special case of a permutation matrix aka an exchange matrix. It is clarified in the following example:

$$V = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \underbrace{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}}_W \cdot \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 4 & 5 & 6 \\ 1 & 2 & 3 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

The structure of eqn (9.5) clearly indicates that convolution is a **linear operation**. The expression describe how to calculate each point/pixel of convolution. Since it is a linear operation, it must be possible to represent it in an equivalent matrix form. It can be shown that the convolution is a matrix product of specially formed block-matrices, the **left convolution matrix** of order  $R$ , and the **right convolution matrix** of order  $c$ ,  $\text{rcm}_c$ , where  $R$  is a number of rows in  $X$  and  $c$  is number of columns in  $W$ , that is:

$$W \star X = X \star W = \text{lcm}_R(W) \cdot \text{rcm}_c(X) = \text{lcm}_r(X) \cdot \text{rcm}_C(W) \quad (9.10)$$

where the convolution matrices are defined as follows:

The **left convolution matrix**  $\text{lcm}_R(W)$  is composed of the weight matrix  $W$  replicated in columns  $R$  times, each block shifted down by one row in the following way:

$$\text{lcm}_R(W) = \begin{array}{c} \longleftarrow c \cdot R \longrightarrow \\ \begin{array}{|c|c|c|c|c|} \hline & 0 & 0 & \cdots & 0 \\ \hline W & & 0 & \cdots & 0 \\ \hline & W & & 0 & 0 \\ \hline 0 & & \cdots & W & 0 \\ \hline 0 & 0 & & & W \\ \hline 0 & \cdots & 0 & & \\ \hline 0 & \cdots & 0 & 0 & \\ \hline \end{array} \begin{array}{l} \uparrow \\ R-1 \\ \downarrow \\ \uparrow \\ r \\ \downarrow \end{array} \end{array} \quad (9.11)$$

The size of the left convolution matrix is  $(R + r - 1) \times (c \cdot R)$ .

The **right convolution matrix**,  $\text{rcm}_c(X)$ , consists of one-dimensional convolution matrices of order  $c$  formed from the rows of the image  $X$ :

$$\text{rcm}_c(X) = \begin{array}{c} \leftarrow C+c-1 \longrightarrow \\ \begin{bmatrix} \langle \mathbf{x}_1 \rangle_c \\ \langle \mathbf{x}_2 \rangle_c \\ \vdots \\ \langle \mathbf{x}_R \rangle_c \end{bmatrix} \begin{array}{l} \uparrow \\ R \cdot c \\ \downarrow \end{array} \end{array} \quad (9.12)$$

where  $\mathbf{x}_k$  denotes the  $k$ -th row of the image  $X$ , and the angle brackets denote the 1-D convolution matrix. For a row vector  $\mathbf{x}$  of length  $C$ , the one-dimensional convolution matrix of order  $c$ , also known as the **Sylvester resultant matrix** [?, ?], is formed from the shifted vector  $\mathbf{x}$  in the following way:

$$\langle \mathbf{x} \rangle_c = \begin{array}{c} \begin{bmatrix} x_1 & x_2 & \cdots & x_C & & & \\ & x_1 & x_2 & \cdots & x_C & \mathbf{0} & \\ & & \mathbf{0} & \ddots & \ddots & & \ddots \\ & & & & x_1 & x_2 & \cdots & x_C \end{bmatrix} \begin{array}{l} \uparrow \\ c \\ \downarrow \end{array} \\ \longleftarrow C+c-1 \longrightarrow \end{array} \quad (9.13)$$

Therefore, the right convolution matrix is of size  $(R \cdot c) \times (C + c - 1)$ . After multiplication of left and right convolution matrices the resulting 2-D convolution matrix has the size  $(R + r - 1) \times (C + c - 1)$ .

### 9.2.1 Numerical example

The following numerical example should clarify most of the details

<http://users.monash.edu/~app/Lrn/Mtlb/gpconvEx1.m>

```
X = 11 12 13 14 15
    21 22 23 24 25
    31 32 33 34 35
    41 42 43 44 45
    51 52 53 54 55
    61 62 63 64 65
    71 72 73 74 75
size(X) = [R C] = [7 5]

W = 0.11 0.12
    0.21 0.22
    0.31 0.32
size(W) = [r c] = [3 2]

lcmWR = 0.11 0.12 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
        0.21 0.22 0.11 0.12 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
        0.31 0.32 0.21 0.22 0.11 0.12 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
        0.00 0.00 0.31 0.32 0.21 0.22 0.11 0.12 0.00 0.00 0.00 0.00 0.00 0.00
        0.00 0.00 0.00 0.00 0.31 0.32 0.21 0.22 0.11 0.12 0.00 0.00 0.00 0.00
        0.00 0.00 0.00 0.00 0.00 0.00 0.31 0.32 0.21 0.22 0.11 0.12 0.00 0.00
        0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.31 0.32 0.21 0.22 0.11 0.12
        0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.31 0.32 0.21 0.22
        0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.31 0.32
size(lcmWR) = [R+r-1 c*R] = [9 14]

rcmXc = 11 12 13 14 15 0
        0 11 12 13 14 15
        21 22 23 24 25 0
        0 21 22 23 24 25
        31 32 33 34 35 0
        0 31 32 33 34 35
        41 42 43 44 45 0
        0 41 42 43 44 45
        51 52 53 54 55 0
        0 51 52 53 54 55
        61 62 63 64 65 0
        0 61 62 63 64 65
        71 72 73 74 75 0
        0 71 72 73 74 75
size(rcmXc) = [R*c C+c-1] = [14 6]
```

```

XCW = conv2(X, W) = lcmWR*rcmXc =
  1.21  2.64  2.87  3.10  3.33  1.80
  4.62  9.88 10.54 11.20 11.86  6.30
 11.23 23.72 25.01 26.30 27.59 14.50
 17.53 36.62 37.91 39.20 40.49 21.10
 23.83 49.52 50.81 52.10 53.39 27.70
 30.13 62.42 63.71 65.00 66.29 34.30
 36.43 75.32 76.61 77.90 79.19 40.90
 33.82 69.48 70.54 71.60 72.66 37.30
 22.01 45.04 45.67 46.30 46.93 24.00
size(XCW) = [R+r-1 C+c-1] = [9  6]

```

### 9.2.2 The proof

The proof of eqn (9.10) is inductive over  $R$ .

For  $R = 1$  the image  $X$  is reduced to a row vector  $\mathbf{x}_1$  and from eqn (9.12) we have

$$\text{rcm}_c(X) = \mathbf{x}_1$$

Eqn (9.11) is reduced to

$$\text{lcm}_1(W) = W$$

The convolution of  $W$  and  $X$  is reduced to the well-known 1-D case, namely, to convolution of each row of matrix  $W$  with a row vector  $\mathbf{x}_1$ , and can be written as

$$W \star X = W \cdot \langle \mathbf{x}_1 \rangle_c = W \cdot \begin{bmatrix} x_1 & x_2 & \cdots & x_c & & \\ & x_1 & x_2 & \cdots & x_c & \mathbf{0} \\ & & \mathbf{0} & \ddots & \ddots & \ddots \\ & & & & x_1 & x_2 & \cdots & x_c \end{bmatrix}$$

Assuming now that eqn (9.10) is true for some  $R$ , we will show that it holds for  $R + 1$ . In this case we have:

$$\begin{aligned}
(W \star X)_{R+1} &= \begin{bmatrix} \text{lcm}_R(W) & 0_{R \times c} \\ 0 & W \end{bmatrix} \cdot \begin{bmatrix} \text{rcm}_c(X) \\ \langle \mathbf{x}_{R+1} \rangle_c \end{bmatrix} = \\
&= \begin{bmatrix} \text{lcm}_R(W) \\ 0 \end{bmatrix} \text{rcm}_c(X) + \begin{bmatrix} 0_{R \times c} \\ W \end{bmatrix} \langle \mathbf{x}_{R+1} \rangle_c = \begin{bmatrix} (W \star X)_R \\ 0 \end{bmatrix} + \begin{bmatrix} 0_{R \times c} \\ W \star \mathbf{x}_{R+1} \end{bmatrix} = \begin{bmatrix} (W \star X)_R \\ W \star \mathbf{x}_{R+1} \end{bmatrix}
\end{aligned}$$

which completes the proof.

### 9.3 A basic building block of a convolution Layer

The basic building block of convolutional layers, equivalent to fully-connected neuronal layers presented, among others in sec. 7.5 and earlier in sec. 6.6, see also Figure 9-1, is presented in Figure 9-2

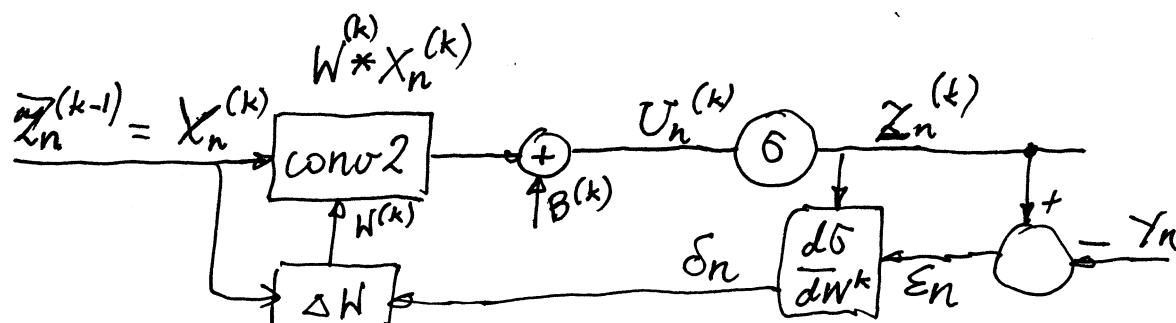


Figure 9-2: A building block of convolutional neural networks

For simplicity of initial explanation we have ignored two facts. The first one is that the  $k$ th convolutional layer consists, in general, of a number, say  $L_k$ , building blocks as in Figure 9-2, hence we work with two set of indexes, e.g.,  $X_n^{(k,l)}$ . As a result the objects are three-dimensional tensors represented by three dimensional arrays. The second simplification is that the entry to each building block  $X_n^{(k,l)}$  is not a matrix, but a  $L_{k-1}$ -dimensional tensor/array. Therefore, we use a P-dimensional convolution as in eqn (9.4). Finally, keep in mind the sample index  $n = 1 \dots N$ . Easy to get lost.