TEKNILLINEN KORKEAKOULU
Tietotekniikan osasto
Teknillisen fysiikan laitos

Harri Lappalainen

A computationally efficient algorithm for finding sparse codes

Diplomi-insinöörin tutkintoa varten tarkastettavaksi jätetty diplomityö.
Työn valvoja: Professori Erkki Oja
Työn ohjaaja: Dosentti Juha Karhunen

Espoossa 7.5.1996.

TEKNILLINEN KORKEAKOULU **DIPLOMITYÖN TIIVISTELMÄ**

Tekijä:	Harri Lappalainen	
Työn nimi:	Laskennallisesti tehokas algoritmi harvakoodien etsimiseen	
Title in English:	A computationally efficient algorithm for finding sparse codes	
Päivämäärä: 7.5.1996		**Sivumäärä:** 57
Osasto:	Tietotekniikan osasto	**Professuuri:**
Laitos:	Teknillisen fysiikan laitos	Tik-61 Informaatiotekniikka
Työn valvoja:	Professori Erkki Oja	
Työn ohjaaja:	Dosentti Juha Karhunen	

Tärkeä näkökohta missä tahansa tiedonkäsittelyssä on tapa, jolla tieto esitetään. Jos esitykseen käytetään yksiköitä (neuroneita), jotka voivat olla joko "aktiivisia" tai "passiivisia", tärkeä esityksen tunnusluku on *aktiivisuussuhde*: niiden neuronien osuus, jotka ovat yhdellä kertaa aktiivisia. Alimmillaan luku on *lokaaleissa esityksissä*, joissa kunkin tilan esityksessä suuresta joukosta neuroneita vain yksi on aktiivinen ja loput ovat passiivisia. *Tiheissä koodeissa* jokaisen tilan esityksessä keskimäärin noin puolet yksiköistä on aktiivisia. Koodeja, joissa aktiivisuussuhde on pieni, kutsutaan harvakoodeiksi.

Aktiivisuussuhde vaikuttaa tiedonkäsittelyssä moniin asioihin kuten verkon arkkitehtuuriin ja vikasietoisuuteen, niiden tilojen lukumäärään, jotka voidaan esittää ja tallentaa, yleistyskykyyn sekä oppimisnopeuteen ja -sääntöihin. Harvakoodeissa yhdistyvät lokaalien ja tiheiden koodien hyvät puolet samalla, kun suurin osa huonoista puolista vältetään.

Tässä työssä on esitetty uusi tehokas algoritmi harvakoodien etsimiseen. Laskennallinen kompleksisuus on lineaarisesti verrannollinen neuronien määrään. Tämä poikkeaa aikaisemmista algoritmeista, joissa kompleksisuus on yleensä ollut verrannollinen neuronien määrän neliöön. Työssä tehdyt simulaatiot keinotekoisella ja luonnollisella datalla osoittavat, että algoritmi kykenee löytämään datasta merkityksellisiä piirteitä.

Avainsanat: harvakoodaus, neuraalilaskenta, ohjaamaton oppiminen.

HELSINKI UNIVERSITY OF TECHNOLOGY **ABSTRACT OF THE MASTER'S THESIS**

Author:	Harri Lappalainen	
Title of the thesis:	A computationally efficient algorithm for finding sparse codes	
Title in Finnish:	Laskennallisesti tehokas algoritmi harvakoodien etsimiseen	
Date: 7.5.1996		**Number of pages:** 57
Faculty:	Information technology	**Professorship:**
Department:	Technical physics	Tik-61 Information sciences
Supervisor:	Professor Erkki Oja	
Instructor:	Docent Juha Karhunen	

An important aspect of any kind of information processing is the way in which the information is represented. If we are using representational units, neurons, which can be either "active" or "passive", then an important characteristic of the code is the *activity ratio*, the fraction of active neurons at any one time. At its lowest value it is *local representation*, where each state is represented by a single active unit from a pool in which all other units are silent. In *dense distributed representation*, each state is represented on average by about half of the units being active. Codes with low activity ratios are called *sparse codes*.

The activity ratio affects several aspects of information processing such as the architecture and robustness of networks, the number of distinct states that can be represented and stored, generalisation properties, and the speed and rules of learning. Sparse codes combine advantages of local and dense codes while avoiding most of their drawbacks.

In this work, a new efficient algorithm for finding sparse codes is presented. The computational complexity depends linearly on the number of neurons in the network, as opposed to previous algorithms, where the complexity has typically been quadratically dependent on the number of neurons. The ability of the algorithm to find meaningful features has been demonstrated in simulations with artificial and natural data.

Keywords: sparse coding, neural networks, unsupervised learning.

Preface

This study was carried out in the Laboratory of Computer and Information Sciences at the Helsinki University of Technology.

I wish to thank my supervisor professor Erkki Oja for the opportunity to work in the laboratory and to do my Master's Thesis on this topic. I got a lot of valuable guidance from him and from my instructor docent Juha Karhunen. Their suggestions made the presentation in this work much more clear and also mathematically more rigorous.

I spent one month in Italy writing the manuscript of this thesis. Many thanks to my girlfriend Josefin Frangione for supporting my work and providing a peaceful environment for writing. Special thanks also to the personnel of the Laboratory of Computer and Information Sciences and the Neural Networks Research Centre for sharing daily conversations and helping in all kind of problems.

Otaniemi, May 5, 1996

Harri Lappalainen

Contents

List of notations

The main symbols and abbreviations used in this work are described below. If some symbols are used for other purposes, the meaning is explained in the context.

$\boldsymbol{x}$	input vector
$\hat{\boldsymbol{x}}$	reconstruction of the input vector
$\tilde{\boldsymbol{x}}$	reconstruction error
$\mathcal{E}$	reconstruction error function
$\boldsymbol{w}_i$	weight vector of the neuron i
p_i	projection of the input on the weight vector of the neuron i
r_{ij}	winning ratio between neurons i and j
m_i	preliminary winning strength of the neuron i
μ_i	winning strength of the neuron i
y_i	output of the neuron i
c_{ij}	correlation of the weight vectors between neurons i and j
$\mathbf{W}$	set of winners
h_{ij}	neighbourhood function
α, β, γ	parameters
MLP	multilayer perceptron
PCA	principal component analysis
SOM	self-organising map
VQ	vector quantisation

Chapter 1

Introduction

In everyday life people solve extremely complex information processing tasks without even noticing it. The brain handles the vast amount of sensory information with such ease that it is sometimes difficult to recognise the difficulty of the task. When people started making intelligent computer programmes they concentrated on what they thought to be the highest brain function: logical reasoning. It was assumed that sensory information could be quite easily processed to a form, which could be manipulated using logical operations. When people tried to make programmes able to process sensory information — for example recognise objects from pictures or words from speech — they soon realised that the very first levels of information processing are far from trivial.

Even the simplest animals can solve many difficult sensory processing problems much better than our programmes. No man-made machine has yet been able to outperform even a simple fly, not to talk about animals with more complex nervous system like fish or frog. It is unthinkable to reach the capacity of the human brain in the near future.

The amount of calculations performed by the fastest contemporary computers does, in fact, compare to that of the nervous system of a fly. Still we are pitifully far from building machines that could work in natural environment as well as the flies. So it is not that much the speed that is lacking but knowledge about how the calculations should be done. It seems reasonable to hope that studying the information processing principles underlying the capacity of biological brain would teach us how to built more intelligent computer programmes, and naturally it would be interesting per se to know more about the brain.

There are many approaches to the brain study, e.g. psychological and anatomical studies, biophysical studies of neurons, measurements of neural activity in behaving animals, information theory, and simulations with artificial neural networks. Ideally all of these lines of study support each other, but they have justification to exist irrespective of the others.

The artificial neural networks can model the brain with very different levels of abstraction. Some models describe the physical and chemical processes of individual neurons, while others model only the abstract information processing principles. In this work, I have adopted the latter approach. I have developed a computationally efficient method for learning so called sparse codes.

1.1 Sparse coding

An important aspect of any kind of information processing is the way in which the information is represented. In the brain it could be represented by the activity of single, individually meaningful neurons, or it could be that only the global activity pattern across a whole neuron population corresponds to interpretable states. There are strong theoretical reasons and experimental evidence (Földiák and Young, 1995; Thorpe, 1995) suggesting that the brain adopts a compromise between these extremes which is often referred to as *sparse coding*.

Let us suppose that there are binary coding units, neurons, which can be either "active" or "passive". An important characteristic of such a code is the *activity ratio*, the fraction of active neurons at any one time. At its lowest value it is *local representation*, where each state is represented by a single active unit from a pool in which all other units are silent, somewhat like the letters on a typewriter keyboard that are locally encoded. In *dense distributed representation*, each state is represented on average by about half of the units being active. Examples of this are the binary (ASCII) encoding of characters used in computers or the coding of visual images by the retinal photoreceptor array. Codes with low activity ratios are called *sparse codes*.

When the neurons have graded outputs it is more difficult to give a simple measure of sparseness. One is the entropy of the outputs(Attneave, 1954; Barlow, 1960; Watanabe, 1981), which decreases as the outputs become sparser. It is difficult to measure the entropy directly, but in some cases the kurtosis[1] of the outputs can be used instead (Field, 1994). The sparser the outputs, the larger the kurtosis. Although sparse codes do have large kurtosis, the opposite does not necessarily hold. It is possible to construct dense codes with large kurtosis (Baddeley, 1996). In this work, a somewhat heuristic definition of sparseness for graded outputs is used. The code is said to be sparse if a small fraction of the inputs convey most of the information.

The activity ratio affects several aspects of information processing such as the ar-

[1]Kurtosis (K) is the fourth-cumulant counterpart of variance. If variance is defined as $\sigma^2 = E\{(x - E\{x\})^2\}$, then $K = E\{(x - E\{x\})^4\}/\sigma^4 - 3$ is the normalised kurtosis. It gives a measure of peakedness of a distribution. For Gaussian distribution it is zero.

	Local	Sparse	Dense
Representational capacity	Very low	High	Very High
Memory capacity	Limited	High	Low
Speed of learning	Very fast	Fast	Slow
Generalisation	None	Good	Good
Interference	None	Controlled	Strong
Fault tolerance	None	High	High
Simultaneous items	Unlimited	Several	One

Table 1.1: Properties of coding schemes according to Földiák and Young (1995).

chitecture and robustness of networks, the number of distinct states that can be represented and stored, generalisation properties, and the speed and rules of learning (Table 1.1).

The representational capacity of local codes is small: they can represent only as many states as the number of units in the pool, which is insufficient for any but the most trivial tasks. Even when the number of units is as high as that in primate cortex[2], the number of discriminable states well exceeds this number. Making associations between a locally encoded item and an output, however, is easy and fast. Single-layer networks can learn any output association in a single trial by local, Hebbian strengthening of connections between active representation and output units, and the linear separability problem does not arise. In such a *lookup table*, there is no interference between associations to other discriminable states, and learning information about new states does not interfere with old associations. This, however, also means that there will be no generalisation to other discriminable states — which is a fundamental flaw, as we can expect a system never to experience precisely the same pattern of stimulation twice.

Dense distributed codes, on the other hand, can represent a very high number ($\sim 2^N$, where N is the number of units) of different states by combinatorial use of units. They are best suited for minimising the number of neurons needed to represent information. If the number of available neurons is high enough, the representational power is largely superfluous, as the number of patterns ever experienced by the system will never approach the available capacity, and therefore dense codes usually have high statistical redundancy. The price to pay for the potential (but unused) high information content of each pattern is that the number of such patterns that an associative memory can store is unnecessarily low. The mapping between a dense representation and an output can be complex (a linearly nonseparable function), therefore requiring multilayer networks and learning algorithms that are hard to implement biologically. Even efficient supervised algorithms are prohibitively slow,

[2]Cortex is the surface of the brain. Cerebral cortex is responsible for higher brain functions, e.g. sensory processing and planning of motor commands.

requiring many training trials and large amounts of the kind of training data that is labelled with either an appropriate output or reinforcement. Such data is often too risky, time consuming, or expensive to obtain. Distributed representations in intermediate layers of such networks ensure a kind of automatic generalisation (Hinton et al., 1986). However, this often manifests itself as unwanted interference between patterns. A further serious problem is that no new associations can be added without retraining the network with the complete training set.

Sparse codes combine advantages of local and dense codes while avoiding most of their drawbacks. Codes with small activity ratios can still have sufficiently high representational capacity, while the number of input-output pairs that can be stored in an associative memory is far greater for sparse than for dense patterns (Meunier and Nadal, 1995). This is achieved by decreasing the amount of information in the representation of any individual stored pattern. As a much larger fraction of all input-output functions are linearly separable using sparse coding, a single supervised layer with simple learning rules, following perhaps several unsupervised layers, is more likely to be sufficient for learning target outputs, avoiding problems associated with supervised training in multilayer networks. As generalisation takes place only between overlapping patterns, new associations will not interfere with previous associations to nonoverlapping patterns.

Sparseness can also be defined with respect to components. A scene may be encoded in a distributed representation while, at the same time, object features may be represented locally. The number of simultaneously presented items decreases as activity ratio increases because the addition of active units eventually results in activation of “ghost” subsets, corresponding to items that were not intended to be activated.

1.2 Sparse coding in the cortex

Recordings from the sensory areas of cortex have demonstrated the existence of neurons with remarkably sharply tuned responses. Reports of visual responses selective to complex visual forms such as monkey paws in the early 1970s were dismissed by many as inconclusive, but it is much harder to ignore the increasingly detailed descriptions of neurons selective for faces in the temporal lobe (Thorpe, 1995). Field (1994) has proposed that the goal of sensory coding is to produce sparser and sparser representation as the processing proceeds to higher cortical areas.

Evaluating the sparseness of coding in the brain is difficult: it is hard to record a set of neurons simultaneously across which sparseness could be measured. We have more information about neurons’ breadth of tuning across various stimulus sets than about sparseness per se. Coding across stimuli and across cells are, however, closely related. The most immediate observation during physiological experiments is the difficulty of finding effective stimuli for neurons in most cortical areas. Each neuron appears to have specific response properties, typically being tuned to several stimulus parameters. In primary visual cortex, many neurons only respond strongly when an elongated stimulus, such as a line, edge, or grating, is presented within a small part of the visual field, and then only if other parameters, including orientation, spatial frequency (width), stereoscopic disparity, and perhaps colour or length fall within a fairly narrow range. This suggests that at any moment during the animal's life, only a small fraction of these neurons will be strongly activated by natural stimuli (Földiák and Young, 1995).

The problem of finding the preferences of cells becomes even more severe in higher visual areas, such as area V4, and especially in inferotemporal cortex. Cells' preferences in inferotemporal cortex are often difficult to account for by reference to simple stimulus features, such as orientation, motion, position, or colour. Cells here show selectivity for complex visual patterns and objects, such as faces, hands, complex geometrical shapes, and fractal patterns (Földiák and Young, 1995).

It has been argued that the brain uses distributed representations because they are tolerant to damage. However, redundancy far smaller than that in dense codes is sufficient to produce robust behaviour. By simply duplicating units with 99% reliability (assuming independent failures), reliability increases to 99.99%. Sparse representations can be even more tolerant to damage than dense ones if high accuracy is required or if the units are highly unreliable.

Arguments supporting dense coding may also be challenged by empirical evidence. Recent studies on Alzheimer's disease (Hodges et al., 1992) suggest that patients may irreversibly lose specificity, and even whole concepts, independently for individual objects. Salzman and Newsom (1994) have recorded movement-sensitive neurons in the middle temporal area of rhesus monkeys. Their data suggested a mechanism in which monkeys chose the direction encoded by the largest signal in the representation of motion direction. This means that the direction was represented by a very small proportion of all direction encoding neurons. Also, in previous studies Newsom, Britten, and Movshon (1989) found out that, when tested with the appropriate stimuli, the most sensitive neurons can rival or even exceed the performance of a psychophysical observer in the same conditions. They recorded the movement-sensitive neurons in the middle temporal area while the monkey was performing a perceptual discrimination and found that responses of some single neurons contained as much information about the stimulus as the animal's behavioural response.

Chapter 2

Previous algorithms for sparse coding

Although sparse coding is recognised as an important characteristic of information processing in the brain, algorithms for finding sparse codes have not yet been studied extensively and only a handful of algorithms have been proposed (Földiák, 1990; Saund, 1995; Olshausen and Field, 1995; Dayan and Zemel, 1995; Hinton and Zemel, 1994). These can be roughly divided into two groups. In the first group there are the algorithms which use some kind of scheme to minimise the reconstruction error, often combined with constraints for the outputs or error functions which promote sparsity. The other group of algorithms is based on minimising the mutual predictability of activities of neurons. The former approach seems to be computationally more efficient — at least with modern computer technology — but the latter seems to be biologically more plausible.

In sparse coding, finding the right units to represent the input is inherently difficult. Suppose we have one thousand neurons. If we want to represent the input with only one active unit at a time, as in local coding, there are only one thousand possibilities, and we can check all of them. If we wanted to use ten units, which would yield a fairly sparse code, there would be roughly 10^{30} different possibilities. It is obvious that we cannot simply go through all possible combinations to check which one would yield the best representation. Therefore one has to use an approximative approach to find the active neurons. Most algorithms have some kind of iterative procedure to calculate the activities.

Suppose we would like a neural network to learn a good representation for its inputs. We would like the learning to be *unsupervised*, which means that the learning is based on the inputs only. This is in contrast with supervised learning, where an external "teacher" is needed to label the data, that is, to tell the desired outputs for each input. The goal for a supervised network is to learn to mimic this given mapping.

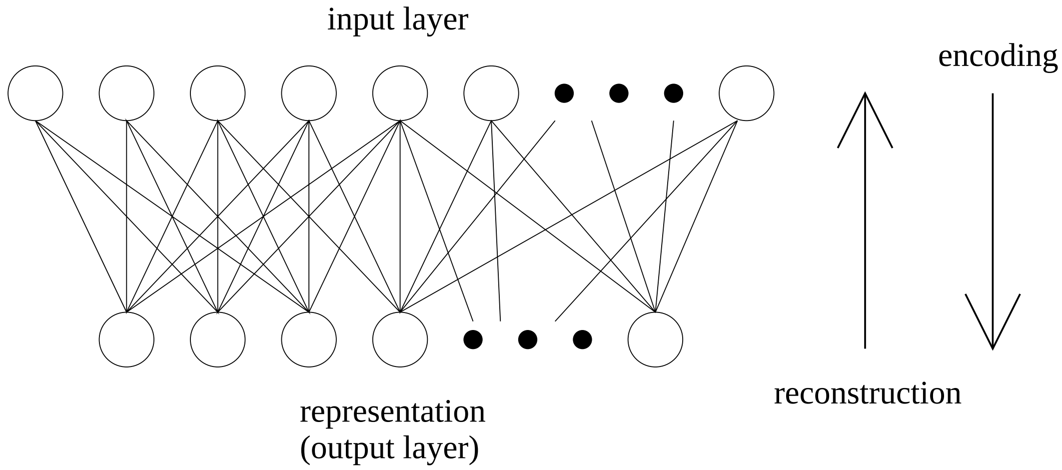


Figure 2.1: Architecture underlying unsupervised learning models based on reconstruction error minimisation.

In order to learn without a teacher an unsupervised network has to have a way to measure the goodness of its mapping by itself. Reconstruction error and mutual predictability are suitable measures because they both guarantee that the outputs contain information about the inputs.

2.1 Reconstruction error minimisation

We shall first consider the algorithms based on reconstruction error minimisation. Let us denote the inputs to the network by $\boldsymbol{x} = [x_1, x_2, \dots, x_n]$, and the outputs of the network by $\boldsymbol{y} = [y_1, y_2, \dots, y_m]$. Then the network can be described by a mapping from the inputs to the outputs: $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$. The reconstruction of the input is then a mapping from the outputs to the inputs $\hat{\boldsymbol{x}} = \boldsymbol{g}(\boldsymbol{y})$. We shall assume that some information is lost and the reconstruction is not exact (cf. pseudoinverse of a matrix). A reasonable way to measure the goodness of the representation $\boldsymbol{y}$ is to measure the reconstruction error $\tilde{\boldsymbol{x}} = \boldsymbol{x} - \hat{\boldsymbol{x}}$ with a given error function $\mathcal{E}(\tilde{\boldsymbol{x}})$. Usually a *quadratic reconstruction error* $||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2$ is used due to its mathematical tractability. Figure 2.1 illustrates the architecture underlying unsupervised learning models based on reconstruction error minimisation.

Due to the information loss in mapping $\boldsymbol{f}$ there is no unique way to choose the reconstruction mapping $\boldsymbol{g}$. This can be illustrated by considering a linear mapping from 2-dimensional input $\boldsymbol{x}$ to 1-dimensional output y. The mapping $y = f(\boldsymbol{x})$ is many-to-one, and therefore it does not have a unique inverse. If we are given the output y, there are many candidate inputs $\boldsymbol{x}$ which could have generated the output, thus we have many possible choices for the reconstruction mapping $\boldsymbol{g}$. There is no obvious way to know which one of them would minimise the reconstruction error. If the reconstruction mapping $\boldsymbol{g}$ is given instead of $\boldsymbol{f}$, however, we can simply define the mapping $\boldsymbol{f}$ to be the one that minimises the reconstruction error. Among all different possible outputs we choose the one that minimises the reconstruction error.

$$\boldsymbol{f}(\boldsymbol{x}) = \arg\min_{\boldsymbol{y}} \mathcal{E}(\boldsymbol{x} - \boldsymbol{g}(\boldsymbol{y})) \tag{2.1}$$

In our example this would mean that we first define a linear reconstruction mapping from y to $\hat{\boldsymbol{x}}$ and then solve the $\boldsymbol{f}$ that minimises the quadratic reconstruction error. It turns out that $f(\boldsymbol{x})$ is a unique linear mapping.

The example shows how it is often difficult to find the reconstruction mapping $\boldsymbol{g}$ for a given mapping $\boldsymbol{f}$, but equation 2.1 makes it easy to find the mapping $\boldsymbol{f}$ from a given reconstruction mapping $\boldsymbol{g}$. By using equation 2.1 it is also easy to impose various constraints on the outputs $\boldsymbol{y}$, because we can restrict the search for the minimum in equation 2.1 to the set of allowed outputs.

A further advantage of using equation 2.1 is that the derivatives of the reconstruction error with respect to the outputs are zero because the outputs minimise the reconstruction error. This greatly simplifies the derivation of the learning rule for the network. If we denote the parameters of the network by $\boldsymbol{w} = [w_1, w_2, \ldots, w_p]$, then, for a given input $\boldsymbol{x}$, the reconstruction error is a function of the outputs and the weights: $\mathcal{E}(\boldsymbol{y}, \boldsymbol{w})$. The goal of the learning is to adapt the parameters so that the average reconstruction error made by the network is minimised. We can derive the adaptation rule for the weights by applying gradient descent. To do this we have to be able to compute the derivatives of $\mathcal{E}$ with respect to the weights w_i.

$$\frac{d\mathcal{E}}{dw_i} = \frac{\partial \mathcal{E}}{\partial w_i} + \sum_j \frac{\partial \mathcal{E}}{\partial y_j}\frac{\partial y_j}{\partial w_i} = \frac{\partial \mathcal{E}}{\partial w_i} \tag{2.2}$$

The last step follows from the fact that in equation 2.1 the outputs are defined to minimise the reconstruction error $\mathcal{E}$ and thus $\frac{\partial \mathcal{E}}{\partial y_j} = 0$ for all j. The last step holds even if there were some constraints for the outputs, which do not depend on the parameters w_i. Let us denote $D_i\boldsymbol{y} = [\partial y_1/\partial w_i, \partial y_2/\partial w_i, \ldots, \partial y_m/\partial w_i]$. This vector points to the direction of the change in the output, when the parameter w_i changes by an infinitesimal amount. If the outputs are constrained, $D_i\boldsymbol{y}$ can only point to a direction where $\boldsymbol{y}$ is allowed to change. The sum in equation 2.2 can be interpreted as the directional derivative of $\mathcal{E}$ in the direction $D_i\boldsymbol{y}$. The outputs minimise $\mathcal{E}$ and therefore the derivative is zero in all allowed directions of change of the outputs.

The outputs could, for example, be restricted to be non-negative. Suppose that the output y_j would be zero due to this constraint. If the output had been allowed to be negative, the reconstruction error could have been further decreased by decreasing y_j. This means that the derivative $\frac{\partial \mathcal{E}}{\partial y_j}$ cannot be zero. This time, however, the output y_j would not change if the parameters changed only by an infinitesimal amount. The output y_j would still remain exactly zero. Therefore, in this example it always holds that either $\frac{\partial y_j}{\partial w_i} = 0$ or $\frac{\partial \mathcal{E}}{\partial y_j} = 0$.

Problems may arise if equation 2.1 has degenerate solutions. If we are using the network in feedforward phase, that is, we are not adapting the parameters, then we can just pick one of the solutions. We might run into problems if we try to adapt the network, because the mapping $\boldsymbol{f}$ has discontinuities where the equation 2.1 has degenerate solutions, and it is not possible to compute the derivatives of outputs y_j in equation 2.2. Luckily in most practical cases the probability of degenerate solutions is zero.

Equations 2.1 and 2.2 provide us a general framework for developing neural networks based on reconstruction error minimisation. The development starts by defining the reconstruction error, the parametrised reconstruction mapping, and possible constraints for the outputs. Sometimes we are able to solve equation 2.1 in a closed form, but in general we have to solve it numerically.

We shall first consider a simple case where we can find the solution to equation 2.1 in a closed form. If we require the reconstruction $\boldsymbol{g}$ to be a linear mapping and there are no constraints imposed on the outputs, it turns out that, assuming the quadratic reconstruction error, also the mapping $\boldsymbol{f}$ will be linear. There are many degrees of freedom in the mapping and we can require the mapping to be orthogonal[1] without any loss of generality. The *principal component analysis* (PCA) gives this kind of orthogonal mapping which is optimal with respect to the quadratic reconstruction error (see e.g. Oja, 1995). The code is maximally dense in the sense that all the neurons participate in representing any given input.

If we require the reconstruction to be a linear mapping and furthermore we require the outputs $\boldsymbol{y}$ to be binary and exactly one neuron to be active at any one time, then we have made sure that the code we obtain is maximally local. The resulting coding scheme is called *vector quantisation* (VQ) (see e.g. Abut, 1990), because we can simply associate each neuron with the corresponding reconstruction and then decide the active neuron by finding the neuron whose reconstruction vector is closest to the input according to equation 2.1[2]. This can be interpreted as quantifying the input.

When the same basic scheme can produce both maximally dense and local codes, it is obvious to try to produce also sparse codes. Saund (1994, 1995) has proposed a few different reconstruction mappings $\boldsymbol{g}$, which can be used for sparse coding. They are chosen so that the resulting coding is sparse whenever the input is sparse, i.e., is describable with a sparse code. Unfortunately the inputs have to be binary, which

[1] For orthogonal matrix W it holds $W^T W = I$. Orthogonal mapping is of form $\boldsymbol{y} = W\boldsymbol{x}$.

[2] If the input falls equally far from two or more reconstruction vectors, then equation 2.1 has degenerate solutions. If the input has a continuous density, however, this happens with probability zero.

restricts the use of the algorithm. It might be possible to generalise the approach for graded inputs, but it does not seem very easy. Olshausen and Field (1995) have developed an algorithm which uses the same kind of linear reconstruction scheme as PCA and VQ, but have added an extra term which promotes sparsity to the reconstruction error function. Equation 2.1 does not have a simple closed form solution in either of these algorithms, and the outputs have to be computed iteratively by gradient descent of the reconstruction error $\mathcal{E}$. This makes the computational complexity depend quadratically on the number of the neurons in the network.

Although the mappings $\boldsymbol{f}$ and $\boldsymbol{g}$ can be related by using equation 2.1, it is also possible to parametrise and learn them both separately. Algorithms using this kind of autoencoder framework have been proposed by several authors, and there has also been modifications which yield sparse codes (Dayan and Zemel, 1995; Zemel, 1993; Hinton and Zemel, 1994). These algorithms can compute the outputs without any time consuming iterations, but unfortunately learning seems to be very slow. This is probably due to the fact that the network has to learn the same things twice as the learning does not take into account the relation between $\boldsymbol{f}$ and $\boldsymbol{g}$ in any way.

2.2 Mutual predictability minimisation

An alternative approach to unsupervised learning of sparse codes is the minimisation of mutual predictability of the outputs of the neurons. In such a scheme, for each neuron there is an associated predictor which tries to predict the output of the neuron on the basis of the outputs of the other neurons (Schmidhuber, 1992). The neurons try to learn a mapping $\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x})$ and the predictors try to learn the mappings $\hat{y}_i = g_i(y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_n)$. The error function for the network is based on the prediction errors $\tilde{\boldsymbol{y}} = \boldsymbol{y} - \hat{\boldsymbol{y}}$. The predictors try to minimise the prediction error and the neurons try to escape the prediction, that is, to maximise the same error criterion. This forces the neurons to represent independent information about the inputs. If a neuron cannot escape the prediction, it is not allowed to be active. When the number of neurons in the network exceeds the number of independent features in the input, this scheme yields a sparse code. Since the neurons try to maximise the error function, care has to be taken to ensure that the outputs are bounded, otherwise it might be possible that the outputs would tend to infinity. In all the algorithms proposed so far, the outputs have been explicitly restricted to a finite range.

One of the first algorithms based on mutual predictability minimisation was published by Földiák (1990). In his model, each neuron computes a weighted sum of its inputs, subtracts the prediction made on the basis of activities of other neurons, and subtracts an individual threshold. The activity of each neuron is then obtained by applying a squashing function. The activity of one neuron depends on the activity of the other neurons, which in turn depend on the activity of the first neuron. Therefore the activities have to be solved iteratively. When a new input is presented to the network, initially all the activities are set to zero. Then the system is iterated until the activities have settled. The final outputs are obtained by rounding the activities to binary values. Sparsity is ensured by introducing a target average activity ratio for the neurons. The individual thresholds are adapted during learning so that the target activity ratios are reached.

Schmidhuber (1992) was probably the first to propose a general framework for mutual predictability minimisation. As an example he used multilayer perceptron (MLP) networks and back-propagation algorithm[3] to teach the neurons and their corresponding predictors. The outputs of the neurons were bounded between zero and one, but due to maximisation of the error criterion the outputs tend to binary values. The strength of the proposed method is its ability to find complex nonlinear dependencies from the data. Moreover, this algorithm does not use an iterative procedure to find the activities of the neurons, but relies on the ability of the nonlinear MLPs to find the correct outputs. However, learning is very slow and the structure is biologically implausible. The binary outputs can also be a disadvantage in some applications. It might be possible to make modifications that allow neurons with graded outputs, but this does not seem straightforward.

Sirosh (1995), and Sirosh and Miikkulainen (1994) have developed a biologically motivated model of cortex which proposes two kinds of lateral connections between neurons. Short range lateral connections are supposed to be excitatory, and long range connections inhibitory. Both types of connections are being adapted during learning. The resulting algorithm is able to model many structures similar to those in the primary visual cortex, such as receptive fields, topographic maps, ocular dominance, orientation and size preference columns, and patterned lateral connections between neurons. While the model gives very interesting results, it is computationally quite demanding. This precludes large-scale simulations which would be important for understanding the massively parallel processing in the cortex.

[3]Back-propagation is a popular learning algorithm for MLP networks. See e.g. Haykin (1994).

Chapter 3

Derivation of the new algorithm

3.1 Basis of the algorithm

In this chapter we derive a new algorithm for finding sparse codes. Emphasis is laid on computational efficiency. The derivation is based on the reconstruction error minimisation scheme, which was discussed in section 2.1. When we are using this scheme, we have to define

1. the reconstruction mapping
2. the reconstruction error function and
3. suitable constraints for the outputs.

Equations 2.1 and 2.2 then provide the framework for derivation of the algorithm. We shall choose a linear reconstruction mapping and quadratic reconstruction error function, because they are mathematically tractable and are able to yield various algorithms depending on the constraints for the outputs. These algorithms have been shown to include versions which yield local, sparse, or dense codes.

We want to choose the constraints so that the resulting code is sparse. We shall formulate a competition mechanism, which is used to impose restrictions to the outputs. The competition mechanism fixes the ratios between the outputs of the neurons, but the overall magnitude is left arbitrary. The minimisation in equation 2.1 is then done over the set of outputs, which has the given ratios between outputs. In other words, the minimisation fixes the overall magnitude of the outputs.

3.2 Competition mechanism

In *competitive learning*, representational units compete for the right to represent the input (Rumelhart and Zipser, 1986). It is unsupervised, which means that there is no need for any external teacher who is labelling the data or telling the learning system the desired outputs. The units, or neurons, which are used in the representation are called *winners*. Both reconstruction error minimisation and mutual predictability minimisation scheme can be interpreted as different methods for implementing the competition.

Hata et al. (1988) studied the visual cortex of cat and found inhibitory connections between neurons responsive to similar stimuli but not between neurons responsive to dissimilar stimuli. This means that neurons sensitive to similar features in the input compete with each other, while neurons sensitive to dissimilar features do not affect each other. This is probably the way in which the brain has implemented sparse coding, and we shall try to mimic this kind of competition behaviour. Our goal is to develop a computationally efficient algorithm which would implement the competition process. It is based on the following principles:

- Neurons sensitive to similar inputs compete with each other
- Neurons sensitive to dissimilar inputs have no mutual competition

Figure 3.1 gives an example of how the competition could work. We have supposed that each neuron has a *weight vector*, which tells the direction of the input where the neuron is most sensitive. When the weight vectors span an orthogonal basis (figure A), the neurons respond to completely dissimilar inputs and there is no sense in finding winners. When there are more weight vectors, however, it is possible to choose a small subset of all neurons to represent the input. This produces more sharply tuned responses and a sparse code (figures B and C). The sharply tuned responses give more explicit information about the input, although the broadly tuned outputs contain the same information. If there are two sets of weight vectors which are tuned to different features of inputs, that is, the weights between sets are orthogonal, then the competition should occur only among neurons in one set. This way the sets could convey independent information and the system could process more information in parallel.

3.2.1 Competition between two neurons

We shall start the derivation of the competition mechanism by considering a network with only one neuron. We shall adopt the reconstruction error minimisation approach, and start by defining the reconstruction mapping (the mapping $\boldsymbol{g}$ in sec-

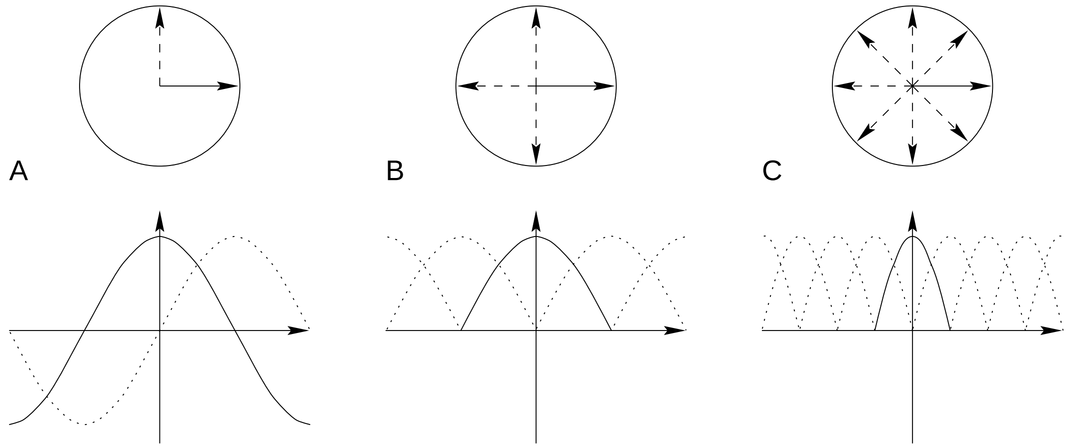


Figure 3.1: The circles show the orientations of weight vectors of neurons. Below the circles are the outputs of neurons for different orientations of inputs. Orientation is on the x-axis and the outputs of the neurons are on the y-axis. Figure A shows the outputs of linear neurons. Two orthogonal neurons span the 2-dimensional input space. In figure B the outputs are limited to be positive. Responses are more localised, and four neurons can represent the 2-dimensional input space. The angles between weight vectors are still at least $90°$, however, and there is no competition. In figure C the neurons are no longer orthogonal and they compete for the input. Competition between similar neurons yields more sharply tuned responses and a sparse code.

tion 2.1), which has parameters $\boldsymbol{w}$:

$$\hat{\boldsymbol{x}} = y\boldsymbol{w}. \tag{3.1}$$

The mapping is linear since the reconstruction is linearly dependent on the parameters $\boldsymbol{w}$ and the output y. Without any loss of generality we can assume that the weight vector $\boldsymbol{w}$ is normalised to unity: $||\boldsymbol{w}|| = 1$. In order to obtain a similar coding as in figure 3.1 B, we shall also require the output y to be non-negative. By assuming a quadratic reconstruction error and solving equation 2.1 (derivation is given in appendix A) we obtain

$$y = \max(0, \boldsymbol{w}^T\boldsymbol{x}). \tag{3.2}$$

We have thus derived a neuron, which is most active when the direction of the input vector is the same as the direction of the weight vector.

Now suppose we have two neurons with reconstruction mapping

$$\hat{\boldsymbol{x}} = \sum_i y_i \boldsymbol{w}_i. \tag{3.3}$$

Again we can assume that the weight vectors $\boldsymbol{w}_i$ are normalised to unity and that the outputs y_i are non-negative. We expect the neurons to compete for the output. All competition should be inhibitory, and therefore we shall require the outputs to be at most what they would be if the neurons were be alone, that is, $y_i \leq \max(0, \boldsymbol{w}_i^T\boldsymbol{x})$. This time the solution to equation 2.1 turns out to be (derivation is given in appendix A)

$$y_1 = \begin{cases} \max\left[0, \min\left(p_1, \frac{p_1 - cp_2}{1-c^2}\right)\right] & p_1, p_2, c > 0 \\ p_1 & p_1 > 0 \text{ and } (p_2 \leq 0 \text{ or } c \leq 0) \\ 0 & p_1 \leq 0 \end{cases}, \tag{3.4}$$

where $p_i = \boldsymbol{w}_i^T\boldsymbol{x}$ and $c = \boldsymbol{w}_1^T\boldsymbol{w}_2$. We have assumed that the degenerate case $\boldsymbol{w}_1 = \boldsymbol{w}_2$, where $c = 1$, can be omitted. The solution for y_2 is similar with indices 1 and 2 interchanged. The value p_i is the projection of the input on the weight vector $\boldsymbol{w}_i$. The value c gives a measure for the similarity of the weight vectors. If $c = 1$, the weight vectors are exactly the same. The smaller the c is the more dissimilar are the weight vectors. We shall call the value c a *correlation*, because in some cases[1] it can be interpreted as the measure of correlation between p_1 and p_2.

It is useful to define a new variable μ and examine its solutions:

$$\mu_i = \begin{cases} y_i/p_i & p_i \neq 0 \\ 0 & p_i = 0 \end{cases}. \tag{3.5}$$

[1] Correlation between p_1 and p_2 is $E\{p_1p_2\} = E\{(\boldsymbol{w}_1^T\boldsymbol{x})(\boldsymbol{w}_2^T\boldsymbol{x})\} = \boldsymbol{w}_1^T E\{\boldsymbol{x}\boldsymbol{x}^T\}\boldsymbol{w}_2$. The value $c = \boldsymbol{w}_1^T\boldsymbol{w}_2$ equals the correlation between p_1 and p_2, if $E\{\boldsymbol{x}\boldsymbol{x}^T\} = I$.

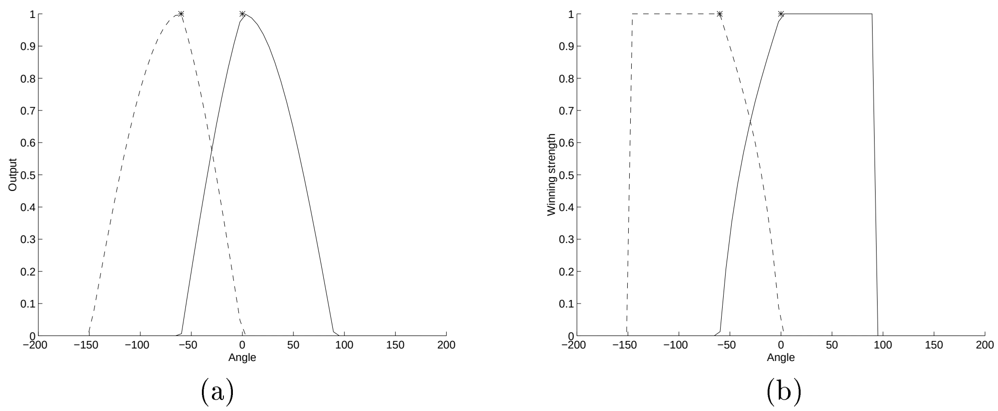


Figure 3.2: The outputs y_i (at the left) and the winning strengths μ_i (at the right) of a network with two neurons are plotted as a function of the angle of the input. The directions of the weight vectors $\boldsymbol{w}_i$ are denoted by '$*$'.

We shall call the new variable a *winning strength*, because it can be interpreted as the result of competition between the neurons. Figure 3.2 shows the outputs y_i and the winning strengths μ_i as a function of the direction of input $\boldsymbol{x}$ (equations 3.4 and 3.5). When only one neuron is active, its winning strength is one. We can interpret this so that the neuron has won the competition and is a full winner. When both neurons participate in the representation of the input, the winning strengths take values between zero and one. Both neurons are winners to some extent, but neither is a full winner. When a neuron has a zero output, also its winning strength is zero, and we can say that the neuron has completely lost the competition.

By examining the equation 3.4 we notice that if the correlation c between neurons is negative or zero, the winning strength is one whenever the projection of the input on the weight vector is positive: $p_i > 0$. This means that neurons with dissimilar weight vectors have no mutual competition. The closer c comes to one, the smaller become the outputs of the neurons. This means that neurons with similar weight vectors compete most. Another notable property of the winning strengths is that they depend only on the direction of the input, not the magnitude. If the input vector $\boldsymbol{x}$ is multiplied with a positive constant, the winning strengths remain unchanged.

3.2.2 Combining the competitions

We could go on formulating reconstruction mappings and solving the output mappings, but this would become increasingly difficult. Moreover, it is not at all clear that we would get a network which would have the kind of competition behaviour we sketched in the beginning of this section. Figure 3.3 shows an example where the simple linear reconstruction does not yield a sparse code. The weight vector

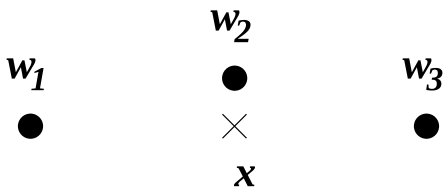


Figure 3.3: An example of a situation where the reconstruction error minimisation does not yield a sparse code. All vectors are supposed to be nearly parallel and 3-dimensional. The figure shows a view from top (cf. a map of a small area on globe). The reconstruction is optimal when neurons 1 and 3 represent the input. However, the code would be more sparse if the neuron 2 represented the input.

$\boldsymbol{w}_2$ is much closer to the input than the two other weight vectors, but according to a linear reconstruction it would not be used in the representation. This is due to the fact that linear reconstruction does not promote sparsity in any way. If we use linear reconstruction and find the optimal outputs using equation 2.1, we obtain a dense code, because the reconstruction error is minimised when all the neurons are allowed to participate in the representation of the input. We need constraints for the outputs which assure the sparsity of the representation.

It should be reminded that we are using reconstruction error minimisation only because it allows easy derivation for algorithms and assures that the representation generated by the network contains information about the input. Our ultimate goal is not to find an exact reconstruction mapping, but to find a mapping which can generate sparse codes. If we can find outputs that approximately minimise the reconstruction error in equation 2.1, we can use the result in equation 2.2 to derive the learning rule and even an approximate minimisation will ensure that the outputs contain information about the inputs.

We shall now leave the reconstruction mapping for a moment and consider a way to formulate the constraints for the outputs. We have already found out that the winning strengths in equation 3.5 are able to measure the competition between two neurons. If we have more than two neurons we can still measure the competition between each pair of neurons and try to combine these competitions to a final winning strength. By combining equations 3.4 and 3.5 we can define a winning ratio r_{ij} between each pair of neurons i and j:

$$r_{ij} = \begin{cases} \max\left[0, \min\left(1, \frac{1-c_{ij}p_j/p_i}{1-c_{ij}^2}\right)\right] & p_i, p_j, c_{ij} > 0 \\ 1 & p_i > 0 \text{ and } (p_j \leq 0 \text{ or } c_{ij} \leq 0) \\ 0 & p_i \leq 0 \end{cases}, \tag{3.6}$$

where $p_i = \boldsymbol{w}_i^T \boldsymbol{x}$ and $c_{ij} = \boldsymbol{w}_i^T \boldsymbol{w}_j$. If there were only two neurons in the network we could write $\mu_1 = r_{12}$ and we would get the same solution as in equation 3.5. When we have more than two neurons we need a way to combine the winning ratios in order to obtain the winning strengths μ_i.

One can think that the winning ratios r_{ij} are fuzzy truth values[2] for propositions "Neuron i is a winner when it competes with neuron j". This suggests the use of some kind of fuzzy and-operation to combine the winning ratios: neuron i is a winner if it wins the competition with neuron 1 and 2 and ... and n. A simple choice for and-operation is the product of truth values. We could simply calculate the product of r_{ij} over index j to obtain the winning strengths for neuron i.

However, there would be a small problem. Suppose that neuron 1 has lost the competition with neuron 2, that is, $r_{12} = 0$. It follows that $\mu_1 = 0$. This is reasonable, since the neuron cannot be a winner if it has totally lost in competition with another neuron. The problem is that r_{i1} for other neurons than 2 may differ from one. This means that other neurons are competing with a neuron that has already lost. Since the lost neuron does not convey any information it would be reasonable that it would not affect the outputs of other neurons. This can be fixed by weighing the winning ratios r_{ij} in the product by the winning strengths. We can now examine the solutions to the *preliminary winning strengths* m_i in the following equation:

$$m_i = \prod_{j \neq i} r_{ij}^{\gamma m_j}. \tag{3.7}$$

We have named the values m_i preliminary winning strengths, since they do not necessarily satisfy the property $y_i = \mu_i p_i$, but we can use m_i to compute values μ_i which do satisfy it, as will be shown later on. The weights are put to the exponent, because we are dealing with a product of numbers. We have also introduced a parameter γ which can be used to control the amount of competition.

The preliminary winning strengths m_i appear in both sides of equation 3.7 and it is impossible to find a solution in a closed form. However, this time we can prune most of the neurons from the computation of m_i. If for some neuron i the winning ratio with another neuron j is zero, that is, $r_{ij} = 0$, then we can omit the neuron i from the computations, since m_i will anyway be zero and it does not affect the m_j of other neurons. This allows us to find a *set of winners*: the set of neurons i, for which the winning ratio $r_{ij} > 0$ for all j. Only these neurons will have positive preliminary winning strengths. For all the other neurons the preliminary winning strength $m = 0$.

[2]In conventional logic propositions can be either true or false. In fuzzy logic there can be propositions with intermediate truth values. Value one means that the proposition is absolutely true and value zero that it is absolutely false. Intermediate values mean that the proposition is true and false to some extent.

3.3 Finding the outputs

We shall now return back to the reconstruction mapping. We derived the winning ratios r_{ij} by starting with two neurons and the reconstruction mapping in equation 3.3. The optimal outputs were then defined by equation 2.1. The relation between the winning strength μ and the output y was defined in equation 3.5 to be $y_i = \mu_i p_i$. The reconstruction mapping was thus

$$\hat{\boldsymbol{x}} = \sum_i y_i \boldsymbol{w}_i = \mu_i p_i \boldsymbol{w}_i. \tag{3.8}$$

There are no guarantees, however, that we could use the m_i defined in equation 3.7 in place of μ_i. There is no reason why the reconstruction defined in equation 3.8 would be even close to the optimal. It turns out, however, that we are already quite close and can get even closer by making a few corrections to m_i. The corrections are such that they approximately preserve the relations between the winning strengths, but correct their overall magnitudes so that we get an agreement with the reconstruction equation 3.8.

Again it should be emphasised that we want to get an agreement with the reconstruction mapping only in order to be able to derive the learning rule and to make sure that the outputs contain information about the inputs. For these purposes even an approximation will suffice.

We would now like to find a mapping from m_i to μ_i. It should be such that equation 3.8 yields a fairly good reconstruction. We shall make two corrections to the preliminary winning strengths m_i. We obtain the first one by considering n neurons, which have nearly parallel weight vectors and which are equally far from the input $\boldsymbol{x}$, that is, $c_{ij} \approx 1$ and $p_i = p_j$ for all i, j. Then it follows from the definition of the winning ratios in equation 3.6 that $r_{ij} \approx 1/2$. By symmetry, the solution to equation 3.7 has to be $m_i = m_j$. From these initial conditions we get

$$m_i = (1/2)^{(n-1)\gamma m_i}. \tag{3.9}$$

On the other hand, we know that for nearly parallel weight vectors the sum of μ_i should be near to one in order to minimise the reconstruction error in equation 2.1 if we are using the reconstruction mapping in equation 3.8. Thus the optimal solution is $\mu_i = 1/n$. Solving $1/n$ from equation 3.9 gives the first correction to m_i.

$$\mu_i' = \frac{\gamma m_i}{\gamma m_i - \log_2 m_i} \tag{3.10}$$

After the first correction the winning strengths are μ_i'. The correction function defined in equation 3.10 is monotonically increasing and it will therefore leave the order of the outputs unchanged although the relations may change.

We get the second correction when we notice that if the μ'_i were optimal in terms of equation 3.8, then p'_i in the following equation

$$p'_i = \boldsymbol{w}_i^T \hat{\boldsymbol{x}} = \sum_j y_j \boldsymbol{w}_i^T \boldsymbol{w}_j = \sum_j \mu'_j p_j c_{ij} \tag{3.11}$$

should equal p_i. If $p'_i \neq p_i$ it means that μ'_j should be multiplied by p_i/p'_i to make the equation 3.8 hold. However, for those neurons j that most contribute to p'_i the correlation c_{ij} is biggest, which means that for them $p_j/p'_j \approx p_i/p'_i$. We shall therefore make the correction by p_i/p'_i to the neuron i only.

$$\mu_i = \max\left[0, \min\left(1, \mu'_i \frac{p_i}{p'_i}\right)\right] \tag{3.12}$$

The equation 3.12 will now give the final corrected winning strengths μ_i. The last corrections to the magnitudes of the winning strengths are not globally uniform, because the ratios p_i/p'_i may vary in different directions. The corrections are locally uniform, however. For neurons with similar weight vectors the ratios are also similar, and therefore the last correction preserves the local relations between winning strengths.

If equations 3.11 and 3.12 were iterated, the solution would approach the optimal solution of equation 3.8. We don't want to do this, however, since the simple linear reconstruction mapping does not promote sparsity in any way. We have obtained a sparse coding by using equation 3.7 and we don't want to lose it. Even a single application of equation 3.12 suffices to bring the outputs to a satisfactory agreement with equation 3.8.

3.4 Finding the set of winners

It was mentioned earlier that the efficiency of the algorithm is based on finding a (small) set of winners, that is, neurons for which the winning strength is not zero, and then calculating the winning strengths for that set only. The set of winners should include those neurons i for which $r_{ij} > 0$ for all j.

In order to find the set of winners we go through all the neurons and at each step update the set. Suppose we have gone through the neurons from 1 to $i-1$ and for those neurons the set of winners is $\mathbf{W}_{i-1}$. Then we start looking at neuron i and update the set of winners to $\mathbf{W}_i$. If $p_i \leq 0$, then r_{ij} is always zero. We don't accept the neuron i to the set, leaving the set intact, that is, $\mathbf{W}_i = \mathbf{W}_{i-1}$. If $p_i > 0$, we start comparing the neuron i pairwise with the neurons $j \in \mathbf{W}_{i-1}$. In each comparison we compute the winning ratios r_{ij} and r_{ji} according to equation 3.6. If $r_{ij} = 0$ we shall discard neuron i and move to the next neuron $i+1$. If $r_{ji} = 0$, we shall remove the neuron j from the set $\mathbf{W}_i$. If $r_{ij} > 0$ for all $j \in \mathbf{W}_{i-1}$ we shall add the neuron i to

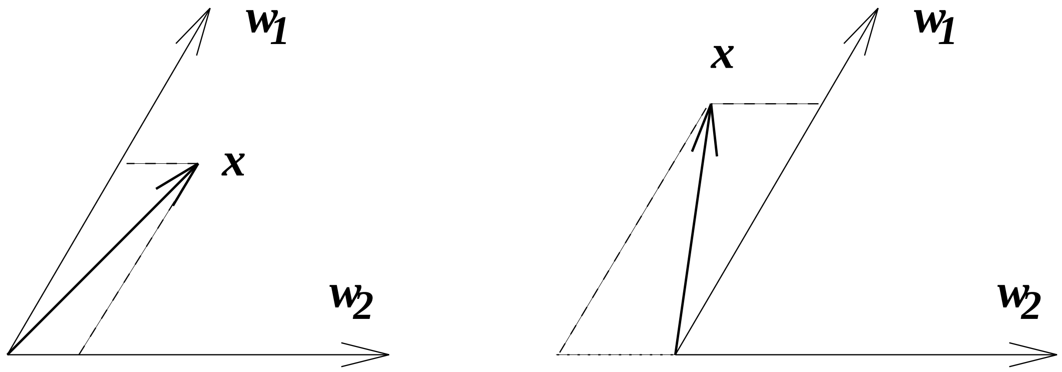


Figure 3.4: At the left, the input is "between" the weight vectors and projection for both neurons is positive. Both neurons are accepted in comparison. At the right, only the projection to neuron 1 is positive. From the point of view of neuron 2, the input is "behind" the neuron 1. Neuron 1 is accepted and neuron 2 discarded in comparison.

the set $\mathbf{W}_i$. Figure 3.4 gives a geometric interpretation for the comparison between two neurons.

The efficiency of this process is due to the fact that at each step a new neuron is compared only with the neurons currently present in the set of winners. A new neuron is not compared with those neurons that have already been discarded from the set of winners. This can, on the other hand, make the resulting set of winners vary with different orders of neurons. This is exemplified in figure 3.5. Whether neuron 3 will be chosen to the set of winners depends on the order of the neurons. In practice, neurons which could have been left out from the set, if the order of neurons had been different, will anyway have a very small winning strength, and thus the final outputs vary only slightly. If the possible variations in the set of winners turn out to be a problem, it is possible to go through the neurons again, and to check whether any neurons in the set of winners should be discarded. This way the neuron 3 in our example would always be discarded.

3.5 Learning rule

The linear reconstruction mapping in equation 3.8 can be used to derive the learning rule as explained in section 2.1. The parameters of the network are the weight vectors $\boldsymbol{w}_i$ and the learning can be derived by gradient descent of the quadratic reconstruction error. The outputs of the network approximately minimise the same error function and therefore the derivatives of the outputs can be ignored as in equation 2.2.

We would like our network to learn sparse codes but, as has been noted before,

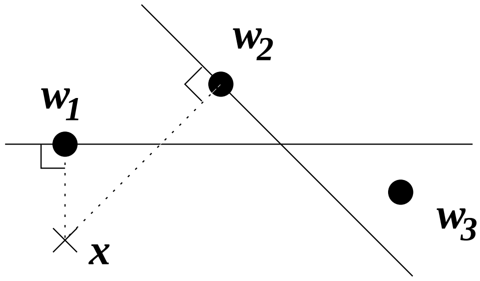


Figure 3.5: An example of a situation where the set of winners varies according to the order of neurons. All vectors are supposed to be nearly parallel and 3-dimensional. The figure shows a view from top (cf. a map of a small area on globe). Neuron 3 is discarded in comparison with neuron 2 and accepted in comparison with neurons 1. Neuron 1 will always be accepted to the set of winners and neuron 2 will always be discarded. Whether neuron 3 is accepted into the set of winners depends on the order of the neurons. If the order is 2, 3 and 1, neuron 3 will be discarded in comparison with neuron 2. If the order is 1, 2 and 3, neuron 2 will be discarded in comparison with neuron 1, and neuron 3 will be accepted in comparison with neuron 1.

linear reconstruction does not promote sparsity in any way. We know that the system will give the best reconstruction for input when all the neurons participate in the representation. If all the correlations $c_{ij} = \boldsymbol{w}_i^T \boldsymbol{w}_j$ between the weight vectors were negative or zero then the neurons would have no competition and they all could participate in the representation. Therefore, if we only minimised the linear reconstruction error defined by equation 3.8, the correlations between weight vectors would decrease until all competition would vanish. Then $p_i > 0$ would automatically mean $\mu_i = 1$, and the code would not be sparse at all.

In order to learn sparse codes, we have to make a minor, biologically plausible addition to the learning rule. The result will be a combination of gradient descent of reconstruction error and use of neighbourhoods as in SOM (Kohonen, 1995). If we only minimised the linear reconstruction error, the weight vectors would be pulled away from each other. The use of neighbourhoods introduces an opposing force which tries to turn the vectors more parallel. Those directions where input density is large get more weight vectors, and the resolution of the representation will be proportionally sharper. This result is qualitatively similar to that with SOM algorithm (Ritter and Schulten, 1986).

We shall first derive the first part of the learning rule by using gradient descent. In order to do this, we have to find the gradient of the quadratic reconstruction error $\mathcal{E} = ||\tilde{\boldsymbol{x}}||^2$ with respect to the weight vectors $\boldsymbol{w}_i$, which are the parameters of the network.

$$\nabla_{\boldsymbol{w}_i}\mathcal{E} = \nabla_{\boldsymbol{w}_i}||\tilde{\boldsymbol{x}}||^2 = \nabla_{\boldsymbol{w}_i}||\boldsymbol{x} - \sum_j y_j \boldsymbol{w}_j||^2 \approx -2\tilde{\boldsymbol{x}}y_i \tag{3.13}$$

Here the reconstruction error $\tilde{\boldsymbol{x}} = \boldsymbol{x} - \hat{\boldsymbol{x}}$. The last step follows from equation 2.2 as explained in the beginning of this section. The standard gradient descent learning rule would be

$$\Delta\boldsymbol{w}_i = \alpha\tilde{\boldsymbol{x}}y_i. \tag{3.14}$$

Here α is the learning rate governing the length of steps taken in the descending direction of the gradient.

The second part of the learning rule will look a bit similar to SOM learning rule (Kohonen, 1995). SOM is a vector quantisation algorithm and is derived using the restriction that exactly one of the outputs is one and the rest are zero. The main difference between SOM and most other VQ algorithms is the use of neighbourhoods in adaptation. In addition to adapting the winning neuron, also its neighbours are adapted. In SOM algorithm, the learning rule for the model vectors $\boldsymbol{m}$ is

$$\Delta\boldsymbol{m}_i = \alpha h_{ic}[\boldsymbol{x} - \boldsymbol{m}_i]. \tag{3.15}$$

The amount of adaptation due to the cooperation is governed by the neighbourhood function[3] h_{ic}. The reconstruction of input for each neuron in SOM is $\hat{\boldsymbol{x}}_i = \boldsymbol{m}_i$. There is also only one winner c and thus $\tilde{\boldsymbol{x}} + \hat{\boldsymbol{x}}_c = \boldsymbol{x}$. Therefore the learning rule could also be written

$$\Delta\boldsymbol{m}_i = \alpha h_{ic}[\tilde{\boldsymbol{x}} + \hat{\boldsymbol{x}}_c - \hat{\boldsymbol{x}}_i]. \tag{3.16}$$

Here the weight vector is adapted in the direction of reconstruction error $\tilde{\boldsymbol{x}}$ plus the direction of the reconstruction $\hat{\boldsymbol{x}}_c$. In our case there can be many winners and for each of them the reconstruction $\hat{\boldsymbol{x}}_c = p_c\boldsymbol{w}_c$ is different.

We can combine the use of neighbourhoods in equation 3.16 and the reconstruction error minimisation in equation 3.14 and sum the adaptations over all winners.

$$\Delta\boldsymbol{w}_i = \sum_{c\in\mathbf{W}} \alpha h_{ic}y_c(\beta\tilde{\boldsymbol{x}} + \hat{\boldsymbol{x}}_c - \hat{\boldsymbol{x}}_i) \tag{3.17}$$

We have introduced a coefficient β, which can be used to regulate the ratio between adaptation due to the reconstruction error minimisation and adaptation due to the cooperation in the neighbourhood. The weight vectors have to be normalised after

[3]The neurons are supposed to lie on a low-dimensional map (cf. the 2-dimensional surface of the cortex). The neighbourhood function h_{ic} is a measure of the cooperation between neurons i and c. The cooperation is strongest for neighbouring neurons and it decreases with the distance between neurons.

each adaptation. Due to the normalisation we can drop away the term $\hat{\boldsymbol{x}}_i$. It has the same direction as $\boldsymbol{w}_i$ and therefore normalisation cancels its affect anyway.

The determination of winning ratios in equation 3.6 required knowledge of the correlations c_{ij} between weight vectors. It would be quite a heavy operation to compute the correlations after each input, and therefore we shall use the batch version of the adaptation algorithm. The adaptations are summed over a set $\mathbf{T}$ of indices in the batch. Also the normalisation of weight vectors can be done after each batch. When using the batch version of the learning rule, the learning rate parameter α can be dropped away.

$$\begin{aligned}\boldsymbol{w}_i'(t) = \textstyle\sum_{\tau\in\mathbf{T}}\sum_{c\in\mathbf{W}(\tau)} h_{ic}(\beta\tilde{\boldsymbol{x}}(\tau) + \hat{\boldsymbol{x}}_c(\tau))y_c(\tau) = \\ \textstyle\sum_c h_{ic}\sum_{\tau\in\mathbf{T}}(\beta\tilde{\boldsymbol{x}}(\tau) + \hat{\boldsymbol{x}}_c(\tau))y_c(\tau)\end{aligned} \tag{3.18}$$

$$\boldsymbol{w}_i(t+1) = \frac{\boldsymbol{w}_i'(t)}{||\boldsymbol{w}_i'(t)||} \tag{3.19}$$

Here we supposed that the neighbourhood function h_{ic} does not change in time. As can be seen from the equation, the use of batch version enables us to move the summation over the neighbourhoods outside the batch, and therefore it needs to be done only once after each batch. On the other hand, writing the summations in this order prevents us from adapting only the set of winners and their neighbours. If the batch is sufficiently long, there are so many winners, however, that this order is computationally more efficient.

3.6 Summary of the algorithm

We shall now summarise the algorithm. Figure 3.6 schematically shows the structure of the network. The network has a layer of neurons, which compute projections p of the input $\boldsymbol{x}$ in the direction of their weight vectors $\boldsymbol{w}$.

$$p_i = \boldsymbol{w}_i^T\boldsymbol{x}. \tag{3.20}$$

The weight vectors are normalised to unity.

The efficiency of the algorithm is based on selecting for each input a set of winners. The selection is based on winning ratios r_{ij} which are defined in equation 3.6.

The ratios are combined according to equation 3.7, which has to be solved numerically since the preliminary winning strengths m_i appear in both sides of the equation. On the other hand, the computation needs to be done for the set of winners only. The preliminary winning strengths m_i have been derived with the following competition mechanism in mind: the stronger is the correlation c_{ij} between the weight vectors of neurons i and j the stronger is the competition between them, and if $c_{ij} \leq 0$ there is no mutual competition between neurons i and j.

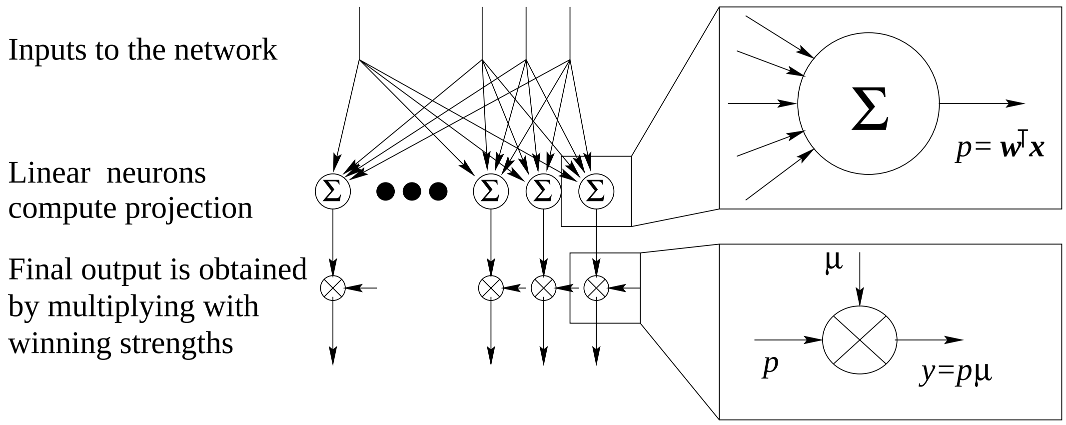


Figure 3.6: The network has a layer of linear neurons and a mechanism which assings winning strengths μ for each neuron.

Equations 3.10, 3.11 and 3.12 are used to compute the final winning strengths μ_i from the preliminary winning strengths m_i. The final outputs y_i are obtained by multiplying the projections with the winning strengths.

$$y_i = \mu_i p_i \tag{3.21}$$

These equations have been chosen so that if we use the following reconstruction mapping for the input

$$\hat{\boldsymbol{x}} = \sum_i y_i \boldsymbol{w}_i,$$

then the reconstruction error is close to minimum.

The learning rule for the network is a combination of minimising the reconstruction error and using neighbourhoods. It is advisable to use the batch version of the learning rule (equation 3.18), because the computation of correlations between the weight vectors ($c_{ij} = \boldsymbol{w}_i^T \boldsymbol{w}_j$) is quite a heavy operation.

3.7 Extensions of the algorithm

In the proposed network, at the first stage the neurons compute projections of the input on their weight vectors, and afterwards only these projections are used to compute the final outputs. Therefore, the network is invariant in orthogonal transformations. If both the input and the weight vectors are rotated with an orthogonal matrix (for which $\mathrm{A}^T\mathrm{A} = \mathrm{I}$), both the projections and the correlations between weight vectors remain unchanged, and thus also the outputs remain unchanged. This property allows us to reduce the dimensionality of the input with a linear PCA network.

If the input is represented in spherical coordinates (r, $\boldsymbol{\phi}$), the winning strengths are functions of $\boldsymbol{\phi}$ only. Thus the outputs are linear functions of r and continuous nonlinear unimodal functions of $\boldsymbol{\phi}$. It is possible to make the outputs depend nonlinearly from the radius r by writing $y_i = f(p_i)\mu_i$ in equation 3.21. This might be desirable if we were to model the responses on the cortex. For example, it has been found that in the primary visual cortex the response of cells to an oriented line is more or less the same whether the line is dim or bright (Sirosh, 1995, page 99). Thus, in our terms, the responses of the neurons depend on the winning strengths only and $f(p)$ saturates to an approximately constant value.

Chapter 4

Evaluation of the algorithm

4.1 Fixed point iteration of winning strengths

The preliminary winning strengths m_i were defined by equation 3.7

$$m_i = \prod_{j \neq i} r_{ij}^{\gamma m_j} = \exp\left\{\sum_{j \neq i} \gamma m_j \ln r_{ij}\right\}.$$

We shall rewrite this

$$\boldsymbol{m} = \boldsymbol{f}(\boldsymbol{m}), \tag{4.1}$$

where $\boldsymbol{m} = [m_1, m_2, \ldots, m_n]$ and $\boldsymbol{f} = [f_1, f_2, \ldots, f_n]$. This equation has to be solved numerically, since m_i appear in both sides of the equation, and it is impossible to find the solution in a closed form. The computation only needs to be done for the neurons in the set of winners, and therefore n equals the number of winners.

Newton's method is often a good choice when one has to numerically solve equations. If we denote $\boldsymbol{F}(\boldsymbol{m}) = \boldsymbol{m} - \boldsymbol{f}(\boldsymbol{m})$, then solving equation 4.1 is equivalent to finding the root of $\boldsymbol{F}$. Finding the root using Newton's method requires the computation of the inverse of the Hessian matrix of $\boldsymbol{F}$, however. It is a n by n matrix, because both vectors $\boldsymbol{m}$ and $\boldsymbol{f}$ have n elements. In some cases it is possible to find an efficient way to compute the inverse of the Hessian. This is possible, if the Hessian is for example close to diagonal matrix. In our case the Hessian is not at all close to diagonal matrix, and it does not seem to have any other properties which would allow us to compute the inverse efficiently.

We are going to solve the equation 4.1 by using another popular method: *fixed point iteration*. If we have an equation of the form $x = f(x)$ we can try to find the solution simply by iterating $x(t+1) = f(x(t))$. This is the standard fixed point iteration. Suppose x_0 is a solution satisfying $x_0 = f(x_0)$. The iteration converges to x_0 only if $|f'(x_0)| < 1$. Notice that if the derivative f' is negative, the value $x(t)$ oscillates around the solution x_0, and if the derivative is positive, the value $x(t)$ changes monotonically. The closer the derivative is to zero the faster is the convergence.

It is also possible to speed up the convergence by modifying the iteration. Aitken extrapolation formula (Atkinson, 1989, page 86) makes the assumption that the values $x(t)$ form a geometric series if we use the standard fixed point iteration. This assumption is usually valid at least when $x(t)$ are close to the solution x_0 and the derivative $f'(x(t))$ is approximately constant. We shall start from $x(t)$ and compute the values $x(t+1)$ and $x(t+2)$ by using the standard fixed point iteration. Then we shall assume that $x(t)$, $x(t+1)$ and $x(t+2)$ are consecutive values in a geometric series which converges to x_0, and solve an estimate $\hat{x_0}$. We can then assign $x(t+3) = \hat{x_0}$ and iterate the process.

If we know that $f'(x) < 0$ for all x, it is also possible to derive an alternative modification which can speed up the convergence. Let us write $g(x, \lambda) = \lambda f(x) + (1-\lambda)x$, and consider the iteration process $x(t+1) = g(x(t), \lambda)$. For a proper choice of λ the derivative $g'(x_0) = 0$, and the convergence is at least quadratic. Even if $f'(x_0) \ll -1$, by choosing λ sufficiently small the iteration can be made to converge (see figure 4.1). We shall find a way to adapt the λ so that the derivative of $g(x, \lambda)$ with respect to x will become as close to zero as possible.

If λ is too large, the derivative is less than zero and $x(t)$ oscillates. This will manifest itself as an alternating sign of the differences of consecutive values of $x(t)$. If λ is too small, the derivative is greater than zero and $x(t)$ changes monotonically. This too can be detected from the sign of the differences, which stays unchanged. We can now formulate a rule for adapting the λ during iteration. After each iteration of $x(t)$ we shall calculate the sign of $x(t) - x(t-1)$. If it has changed after last iteration, we shall decrease λ, otherwise we shall increase it. The use of λ speeds up convergence and allows convergence even if $f'(x_0) < -1$.

In our case the equation 4.1 is a system of equations. We can still use the Aitken extrapolation formula or the adaptation rule for λ proposed above, but this time we have to apply the modifications to each of the functions f_i separately. Both of the modified methods and the standard fixed point iteration were tested in simulations. The iterated function was like that in equation 4.1. There were ten neurons, and the values $\ln r_{ij}$ were sampled from an even distribution between -4 and 0. The closer the values were to zero the better the convergence was. The distribution used in this test can be thought of as a sort of worst case, because normally the values are not this small. Figure 4.2 shows a typical test run. The standard method did not converge at all. The iteration using Aitken extrapolation formula usually converged, but much more slowly than the proposed method, which converged in all tests. The good performance of the proposed method is probably due to the fact that all partial derivatives $\partial f_i / \partial x_j \leq 0$ since $\ln r_{ij} \leq 0$. Therefore the value of each of the variables $m_i(t)$ typically oscillate in the standard fixed point iteration, and the use of adaptive

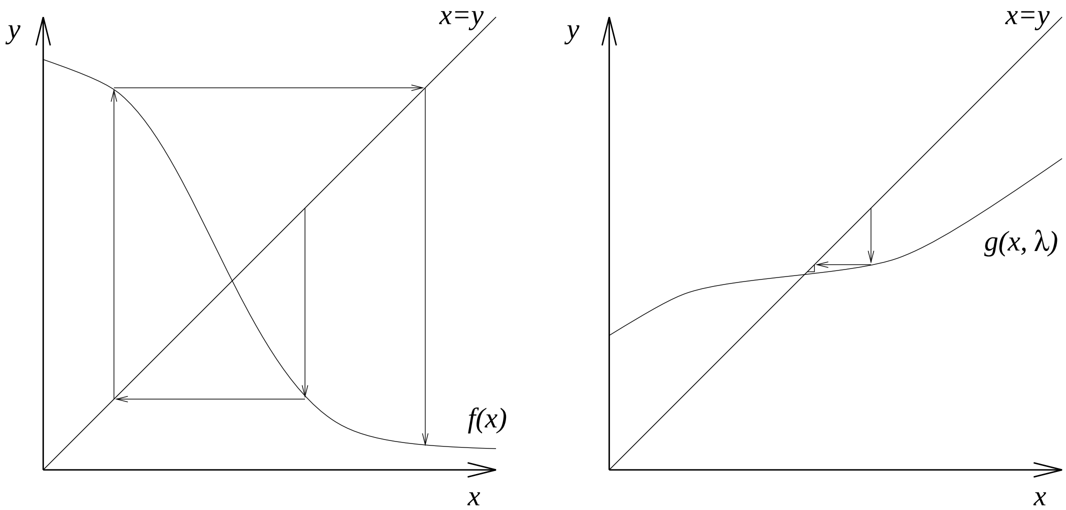


Figure 4.1: In the left, fixed point iteration does not converge to the solution x_0, because $|f'(x_0)| > 1$. The derivative is negative, and therefore the derivative of a new function $g(x, \lambda) = \lambda f(x) + (1 - \lambda)x$ can be made zero by a proper choose of λ. In the right, $\lambda \approx 2/3$, and $g'(x_0, \lambda) \approx 0$.

λ can efficiently dampen the oscillation.

4.2 Computational complexity

Suppose we have a network with N neurons and they all have M inputs. Also suppose that the average size of set of winners is W and we make 10 iterations in the fixed point iteration of m_i. For a single input the computation of p_i takes roughly NM operations. The computation of a_{ij} has to be done for the set of winners only, so it takes order of NW operations. The iteration of m_i takes order of $10W^2$ operations. The computation time of μ_i from m_i is negligible compared the time used in iteration. The adaptation of the weight vectors needs to be done for the set of winners only, and it takes WM operations, which is negligible compared to the term NM. Computation of the correlations c_{ij} between the weight vectors takes N^2M operations, but if the size of batch is much larger than N, this is again negligible.

Summing the amounts of operations gives a total of $NM + NW + 10W^2$ operations. The number of winners depends on the structure of data and generally saturates to a constant value, when the number of neurons grows. It is very rarely greater than the number of inputs. Therefore the first term dominates, and *computational complexity is typically of order NM*. This means that the computational complexity grows linearly proportionally to the number of neurons as opposed to previous algorithms where the growth has been at least quadratic.

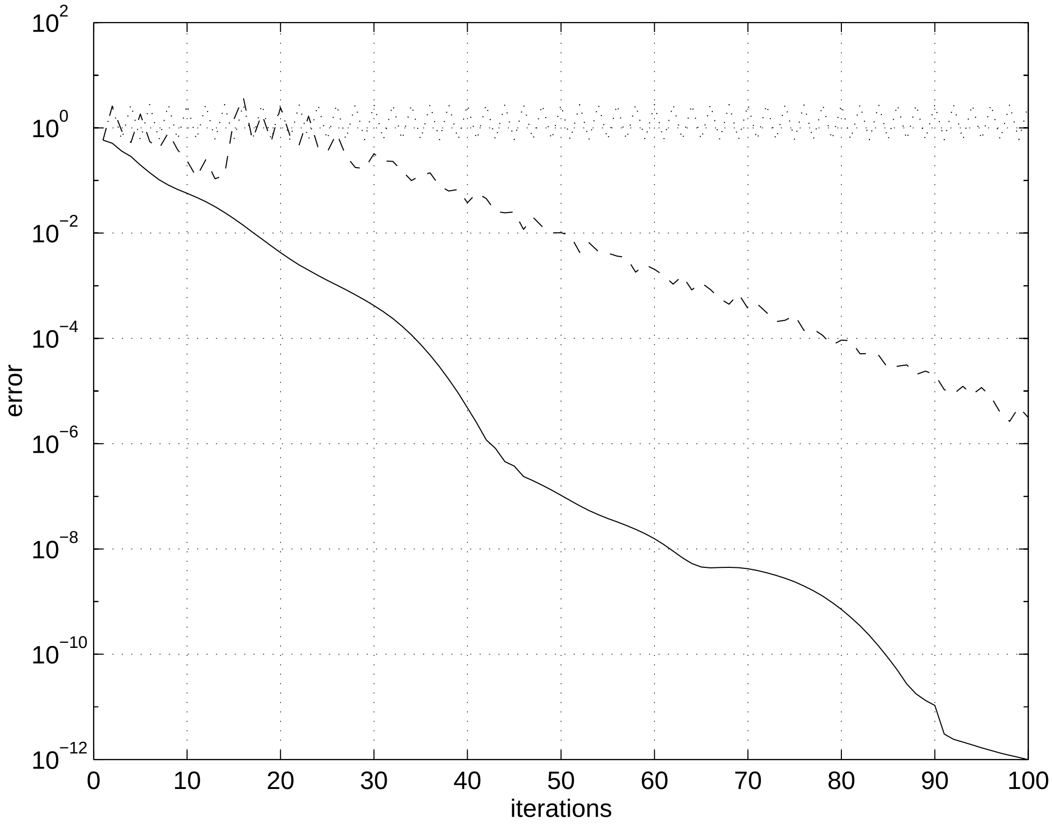


Figure 4.2: The proposed modified fixed point iteration method using adaptive λ (solid line) has been compared with the standard method (dotted line) and Aitken extrapolation formula (dashed line). The proposed method has converged much more rapidly than the iteration using Aitken extrapolation formula, while the standard method has not converged at all.

4.3 Simulations

In this section the ability of the algorithm to learn sparse codes is verified in simulations. Section 4.3.1 shows what are the outputs of the network, when the weight vectors are fixed. The outputs will look very much like those presented in figure 3.1. Sections 4.3.2 and 4.3.3 show that the algorithm is able to learn meaningful sparse representations from artificial and natural data.

4.3.1 The outputs of the network

The neurons compute the projection in the direction of their weight vectors. Therefore the output of a neuron before the winning strength assignment is the cosine of the angle between the input and the weight vector multiplied with the norm of the input vector. The outputs are limited to be positive, and so the output of a single neuron looks like a cut sinusoidal after the winning strength assignment. When there are more than one neuron, they compete for the input. Generally it can be said that neurons compete the stronger with each other the closer their weight vectors are. If there are two sets of neurons which have orthogonal weight vectors, their winning strengths are independent, and therefore the system can convey independent information in parallel.

Figure 4.3 shows the shape of y, when the input is 2-dimensional, and there are three evenly spaced weight vectors. A notable property of μ is that they depend only on the direction of the input, not the amplitude. Therefore the y are linearly dependent on the length of the input. The length of input vector has been normalised to unity, but the shape of the outputs would remain unchanged also for different lengths of input. In this test, the parameter γ, which governs the amount of inhibition, has been one. When the weight vectors are orthogonal there is no competition, and the neurons work completely independently. When the weight vectors become more parallel the amount of competition increases, which sharpens the outputs.

Figure 4.4 shows the effect of parameter γ. The larger the γ the stronger the inhibition. When γ approaches infinity, the winning strengths become binary. From the figure it can be seen that increasing γ makes the transition between winners sharper.

Figure 4.5 shows the shape of y when the input space is 3-dimensional. Weight vectors are nearly parallel and are organised on a 2-dimensional lattice. The figure shows that the properties demonstrated in two dimensions generalise in higher dimensions. The figure shows again how increasing γ makes the transition between winners sharper.

Figure 4.6 shows what happens when there are orthogonal sets of nearly parallel weight vectors. The competition occurs only inside one set and therefore the system

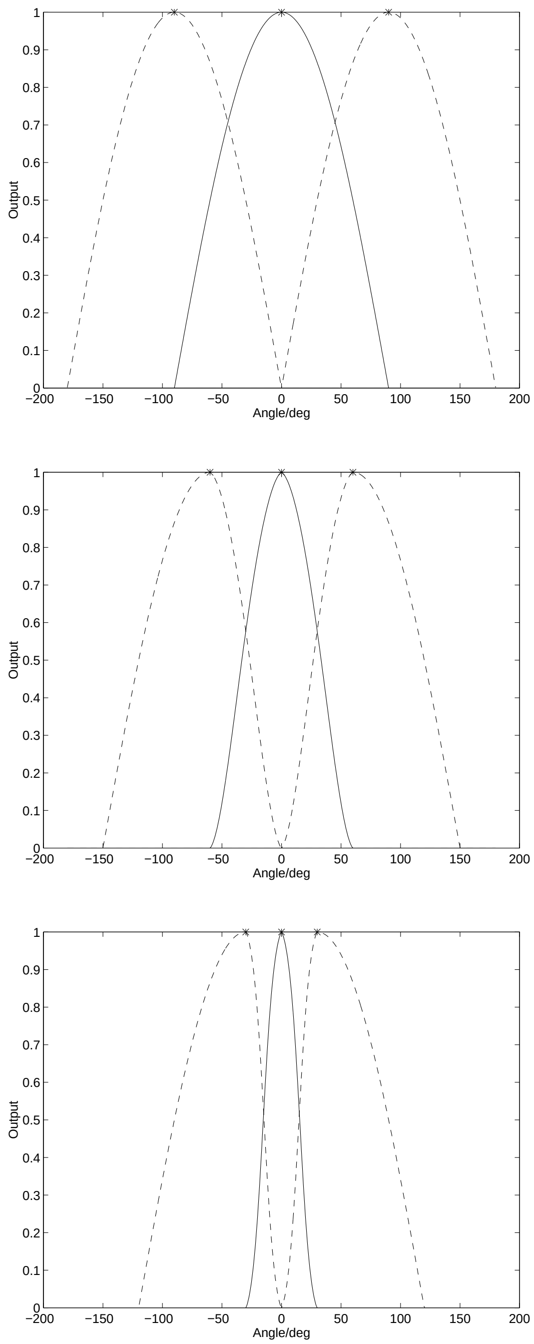


Figure 4.3: The response for the middle neuron becomes more sharply tuned due to competition when the adjacent neurons come close. X-axis shows the angle between the input and the middle neuron. Places of weight vectors are denoted by '$*$'. At the top, the angles between neurons are 90° and there is no competition. In the middle, the angles are 60° and at the bottom 30°. Parameter $\gamma = 1$.

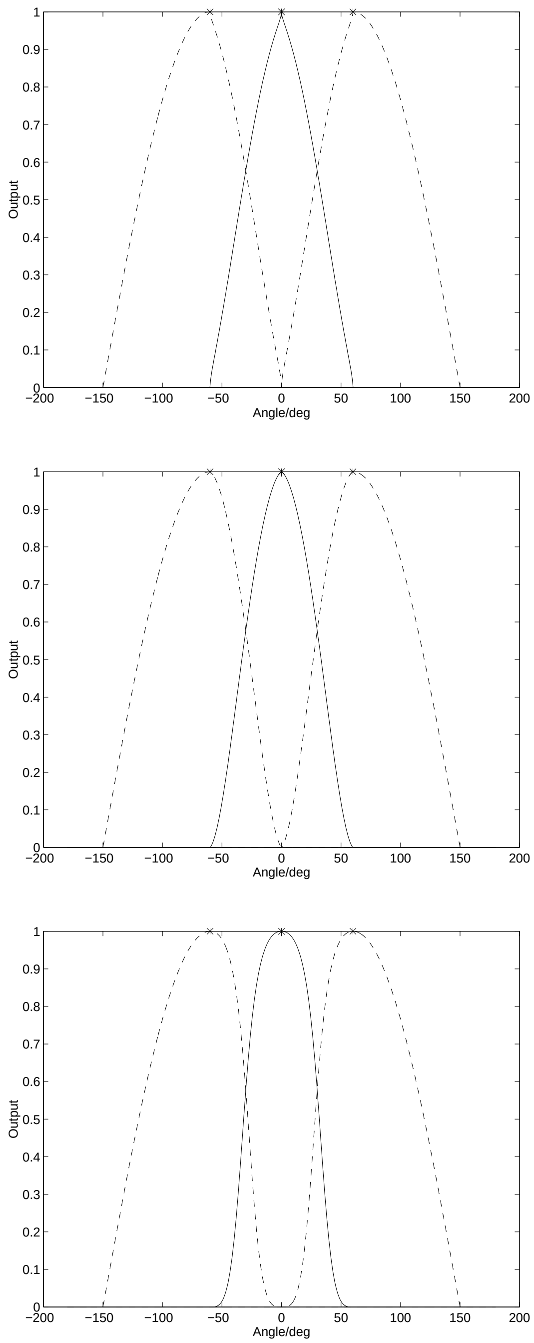


Figure 4.4: The transition between winners becomes sharper as the parameter γ which governs the amount of inhibition increases. At the top $\gamma = 1/3$, at the middle $\gamma = 1$, and at the bottom $\gamma = 3$. The angle between the neurons was $60°$.

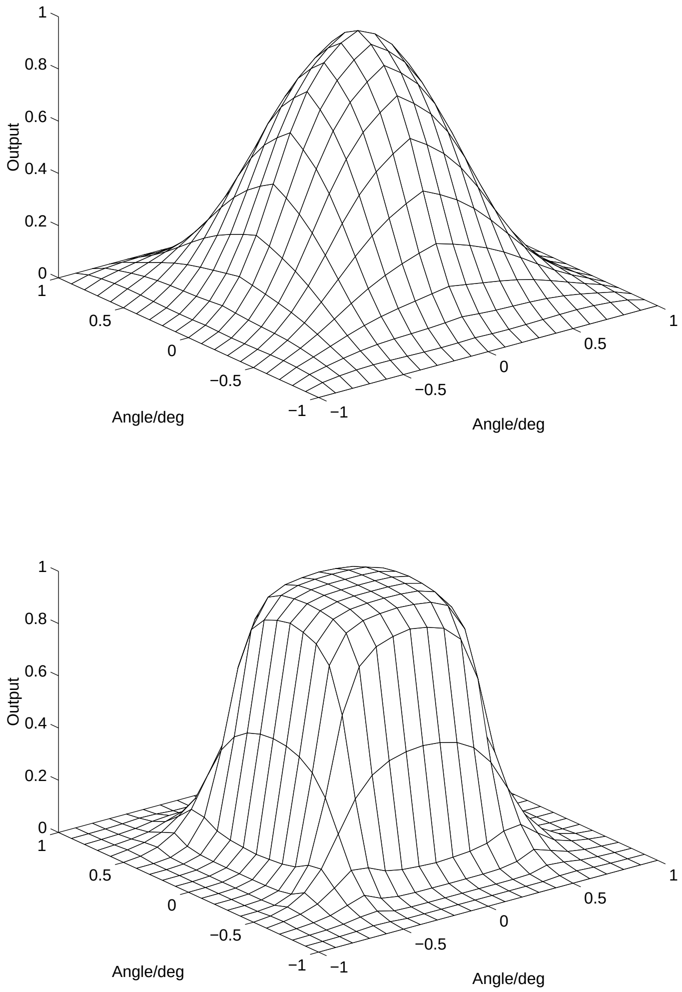


Figure 4.5: Nine nearly parallel weight vectors are organised on a 2-dimensional lattice (angles $-1°$, $0°$ and $1°$ in both x- and y-axis). The plots show the y of the neuron in the middle. At the top, parameter $\gamma = 1$. At the bottom $\gamma = 3$.

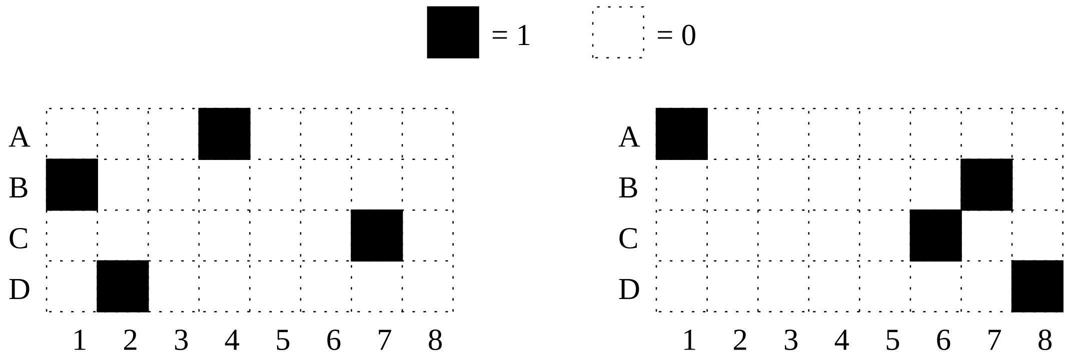


Figure 4.6: Outputs of a network with 32 neurons. There are four sets (A, B, C and D) of eight (1–8) nearly parallel weight vectors. The weight vectors between two different sets are orthogonal, and thus there are four independently functioning parts in the network. The inputs were superpositions of four orthogonal vectors, one weight vector from each set. The outputs show that there is no cross-talk between orthogonal sets. Inside one set the neurons compete with each other, and since the input is a sum of weight vectors the outputs are binary. (If the input had fell between two weight vectors they would have had graded outputs as in figures 4.3–4.5.)

has several independent parts, which enables the system to process information in parallel. The competition inside the sets makes the responses sharply tuned and results in a sparse code.

4.3.2 Learning artificial data

In practice it is usually impossible to divide the network into orthogonal parts. If the network is topologically ordered, the weight vectors of far-off neurons are usually orthogonal, but adjacent neurons are always nearly parallel. Therefore, it is impossible to make a clear division into functionally separate areas. This does not matter since the algorithm does not make any assumptions about functionally separate areas.

The learning algorithm was tested with artificial data, which has this kind of structure. The data was also sparse, which means that it is possible to represent each sample with only a few models from a large pool of models. Olshausen and Field (1995) have conjectured that natural images have this kind of sparse structure. The data can be seen as a strongly simplified model of the visual input to an amphibian. It's environment is assumed to consist of small insects flying around.

Figure 4.10 (a) shows a sample of the data set. There are supposed to be four insects (we shall call them flies) in arbitrary positions in a wrapped 1-dimensional space. (Wrapped means that when you move in one direction, you eventually come to the same place.) We take an imaginary photograph of the flies, but due to the optics, their images appear blurred. In figure 4.10 (a) one can see only three distinct images. This is because two of the flies happen to be in the nearly same position. Each sample can be described by four models of the flies, if the models correspond to the image of a fly in a given position. Since there are infinitely many positions, there may also be infinitely many models. However, the images of two flies in nearly same positions are very similar, and it is possible to discretise the models without making a large error.

In this test, 40 neurons are used. The learning was done in several parts and for each part the learning parameters were constant (table 4.1). The neurons were organised in a 1-dimensional map and the neighbourhood function was piece-wise linear

$$h_{ic} = \begin{cases} 1 & |i-c|-r \leq 0 \\ |i-c|-r & 0 < |i-c|-r < 1 \\ 0 & |i-c|-r \geq 1 \end{cases} \tag{4.2}$$

Here r is the radius of the neighbourhood. Parameters $\gamma = 1$ and $\beta = 1/4$ were held constant throughout the learning.

Number of batches	Number of samples in batch	Neighbourhood radius
10	100	10
10	100	5
20	1000	2
10	1000	1
10	1000	0.5

Table 4.1: Parameters in the test run with artificial data.

The weight vectors learned by the network are presented in figure 4.7. It shows that the network was able to find a sparse representation for the input. Although the flies never occurred alone in the input, each neuron has learned to represent the position of a single fly. The network is topologically ordered: adjacent weight vectors are nearly parallel, while far-off weight vectors are nearly parallel (figure 4.8). Figure 4.9 shows the outputs of the network to inputs with only one fly.

Figure 4.10 shows the outputs to a input with four flies, two of which are in the nearly same position (figure (a)). The projection of the input vector on the weight vectors of the network are shown in figure (b). The competition between neurons with similar weight vectors greatly sharpens the outputs, which are shown in figure (c). The figure clearly shows that the output is a very nonlinear function of the direction of the input vector. Due to the competition between neurons with similar weight vectors the final outputs are much sharper than the projections before competition.

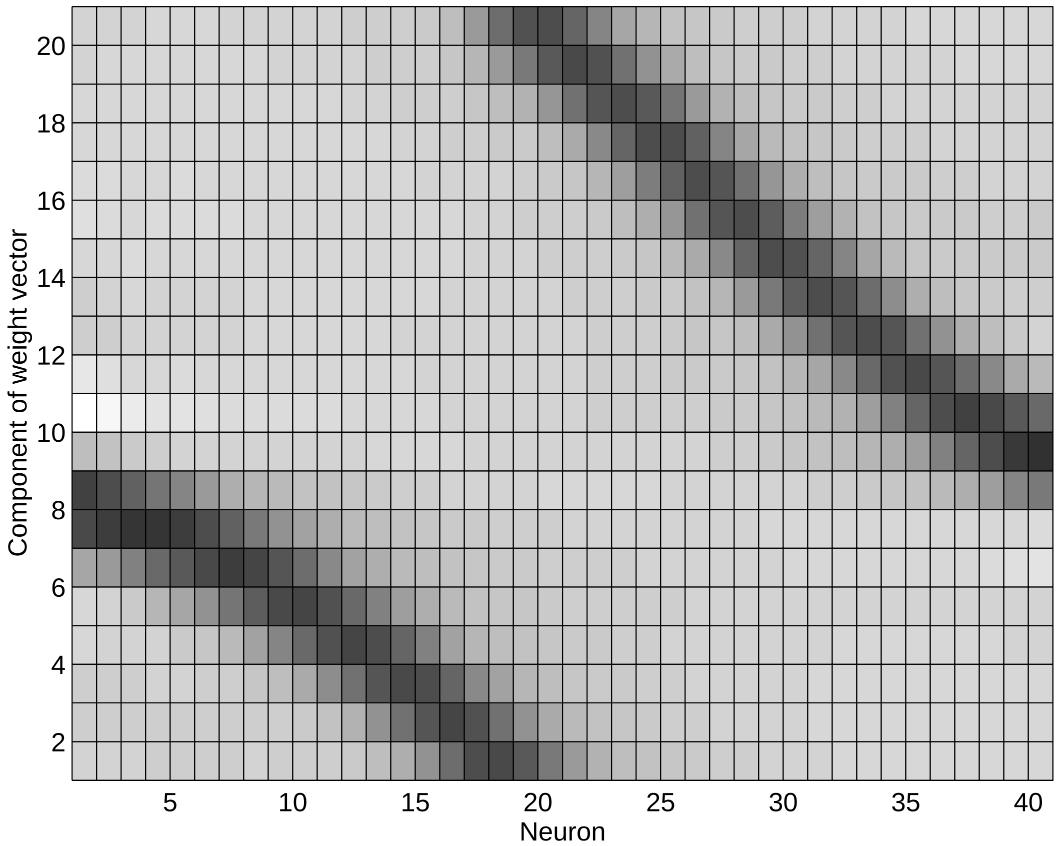


Figure 4.7: The weight vectors of the network. The grayish colour corresponds to zero value. The darker colours are positive values and lighter colour are negative. Most components in the weight vectors are very close to zero. Each neuron has learned to represent the position of a single "fly".

On the other hand, neurons with dissimilar weight vectors do not have mutual competition, which allows different parts of the network to work independently. There are three clear winners among the neurons, which is in agreement with the original model for the input. The network has learned to represent the independent objects generating the input.

4.3.3 Learning natural data

Next the algorithm was tested with natural data. A sheet of hand-written text was digitised with 100 dpi resolution. The samples were obtained by taking small 16 by 16 pixels square areas corresponding to 4mm by 4mm area in the original text. In figure 4.11 (a) there are 100 samples from the image. In the sample vectors, background colour of the sheet was coded by zero and the text was coded with positive

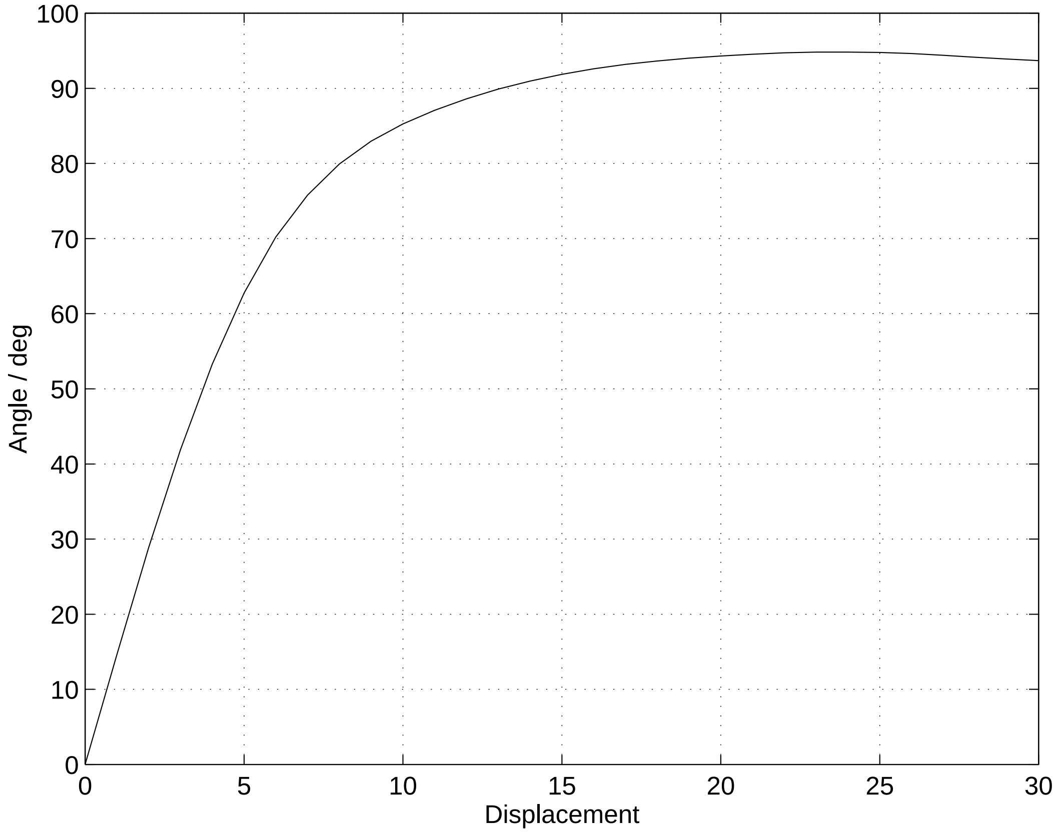


Figure 4.8: The average angle between weight vectors as a function of their distance in the 1-dimensional map.

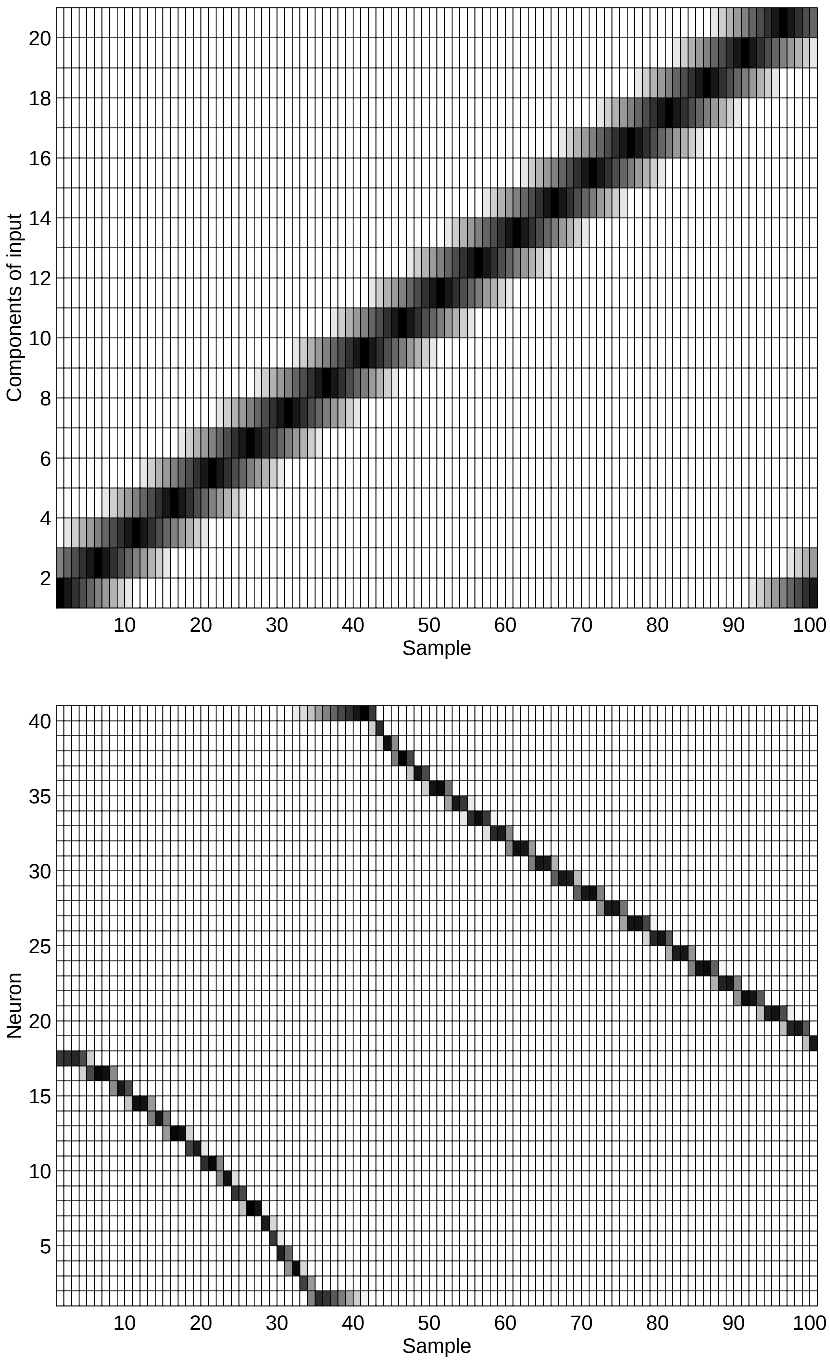
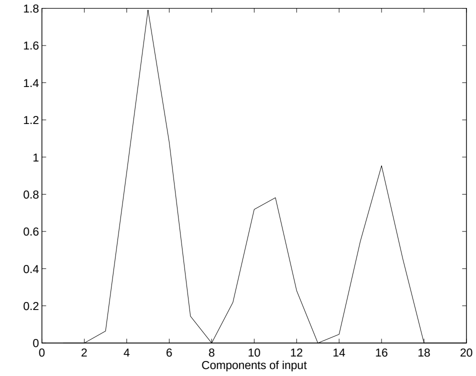
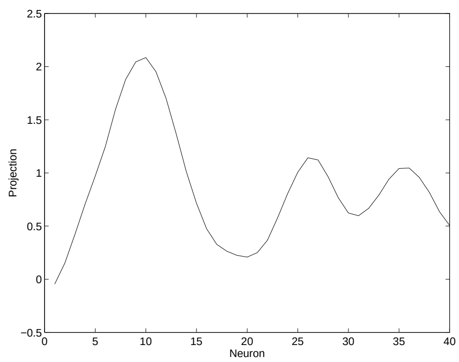


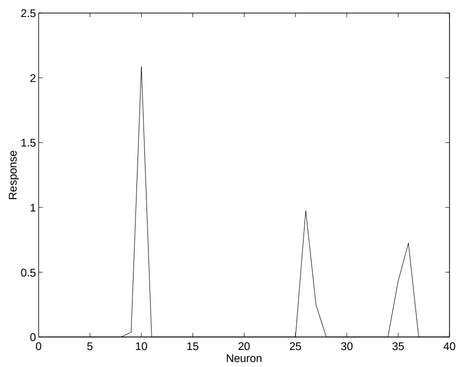
Figure 4.9: The outputs of the map (at the bottom) for sample inputs (at the top). White colour corresponds to zero and dark colours to positive values.



(a)



(b)



(c)

Figure 4.10: A sample from the training set (a), the projections of the neurons (b) and the final outputs after winning strength assignment (c).

values. A 2-dimensional map was adapted using parameters in table 4.2. Parameters $\gamma = 1$ and $\beta = 1/2$ were held constant throughout the learning. Figure 4.11 (b) shows the weight vectors of the resulting map. Each neuron seems to be sensitive to a dot of ink at a localised part of the sample. It is also evident that the finite sampling area has caused the distribution of the neurons to be far from uniform. The neurons which are sensitive to ink at the edge of the sample are much more localised than the neurons which are sensitive to ink at the centre of the sample. This happens probably because there are quite many samples where only a small part of a letter is visible at the edge of the sample.

Number of batches	Number of samples in batch	Neighbourhood radius
10	100	5
10	100	2
10	1000	1
20	5000	0.5
20	5000	0.2

Table 4.2: Parameters in the test runs with 100 neuron maps and natural data.

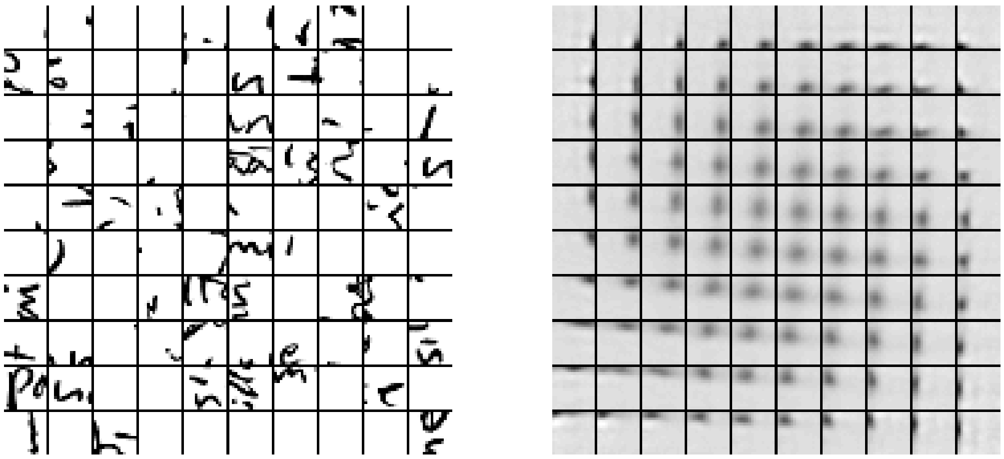
In order to decrease the effect of the finite sampling area, the samples were weighted with a Gaussian function, whose full width at half maximum was the width of the sample, i.e., the weights at the middle of the borders of a sample were 1/2. All parameters were the same as in the previous test. Figure 4.12 shows some of the samples and the resulting map. The centre of the sample received more neurons to represent it. Some neurons have elongated dots in their weight vectors. This means that these neurons are sensitive to location as well as orientation.

Next the effect of increasing the number of neurons was tested. The data was the same as in the previous test but the number of neurons was increased to 400. The parameters used in the training are shown in table 4.3. The resulting map is shown in figure 4.13. Due to the larger number of neurons there is more competition and the neurons become more specialised. The majority of the neurons are sensitive to both location and orientation.

Figure 4.14 shows the linear reconstructions of the sample images from the outputs of the maps. The reconstructions have been computed using equation 3.8. The acuity of the reconstructions based on the map with 400 neurons is naturally better than with 100 neurons. It should be noted that equation 3.8 does not give the exact reconstruction of the inputs, but nevertheless it can be used to give an idea about the amount of information in the outputs of the network. Also, the only goal of the algorithm is not to find optimal representation with respect to reconstruction error, which could be minimised using dense coding. The outputs are also required to be sparse, which has several other advantages over dense coding.

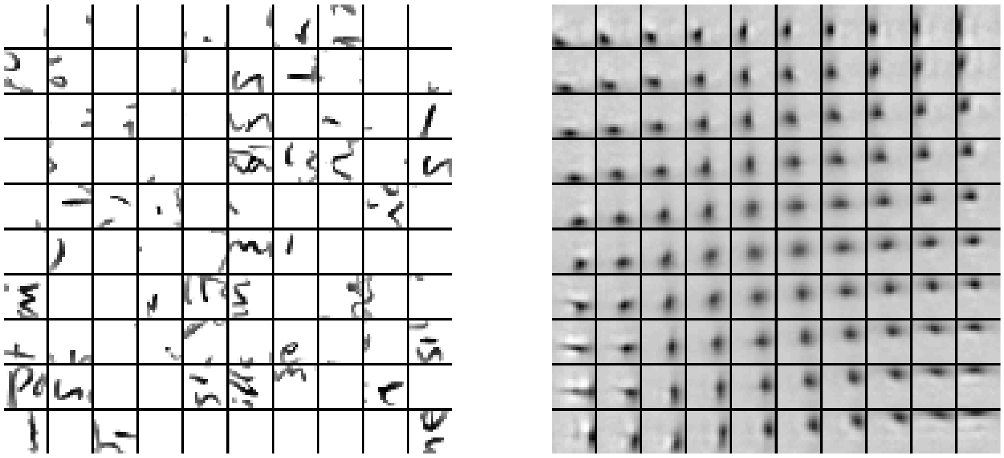
Number of batches	Number of samples in batch	Neighbourhood radius
10	100	10
10	100	5
10	1000	2
20	5000	1
20	5000	0.5
20	5000	0.2

Table 4.3: Parameters in the test run with 400 neuron map and natural data.



(a) (b)

Figure 4.11: 100 samples from hand-written text (a). The text has been digitised with 100 dpi resolution, and each sample has 16 by 16 pixels corresponding to 4mm by 4mm area. White colour corresponds to zero and dark colours to positive values. A 2-dimensional map with 100 neurons (b) has learned to represent the samples. Due to the fairly small area in the samples the distribution of the filters is far from uniform.



(a) (b)

Figure 4.12: 100 samples from hand-written text (a). The samples are otherwise similar to those in figure 4.11 (a), but in order to decrease the effect of the finite sampling area they have been weighted with a Gaussian function, whose full width at half maximum has been the width of the sample, i.e., the weights at the middle of the borders of a sample have been 1/2. A 2-dimensional map with 100 neurons (b) has learned to represent the samples. The distribution of the filters is more uniform than in figure 4.11 (b).

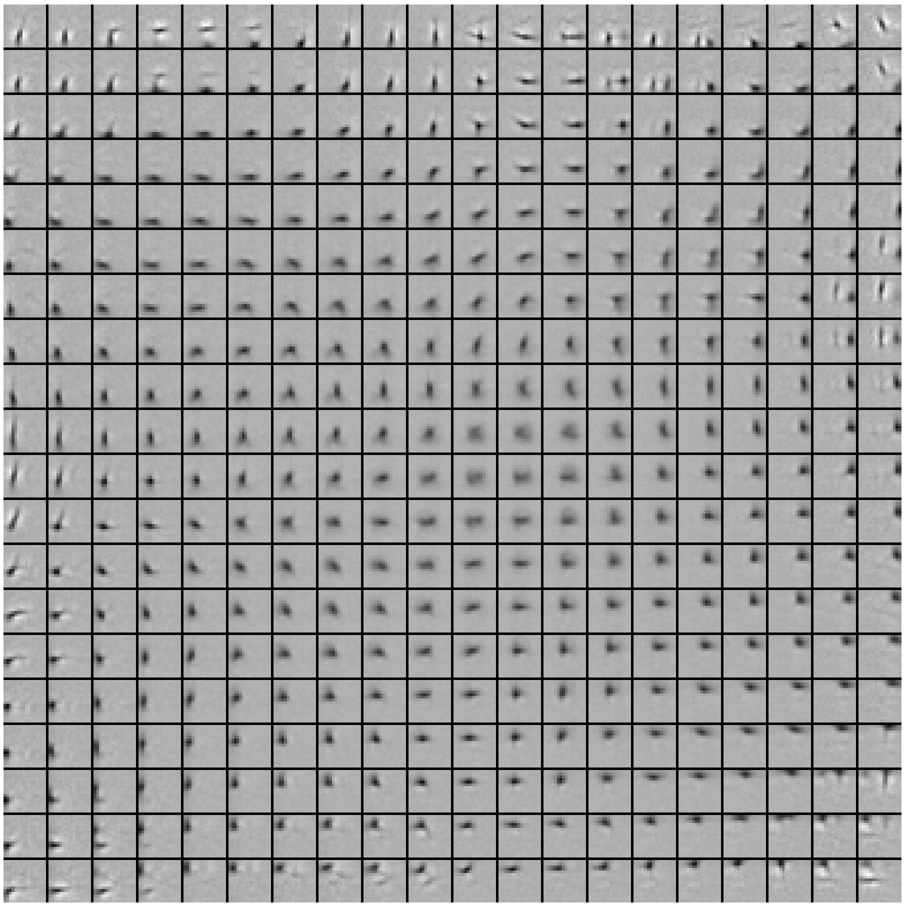
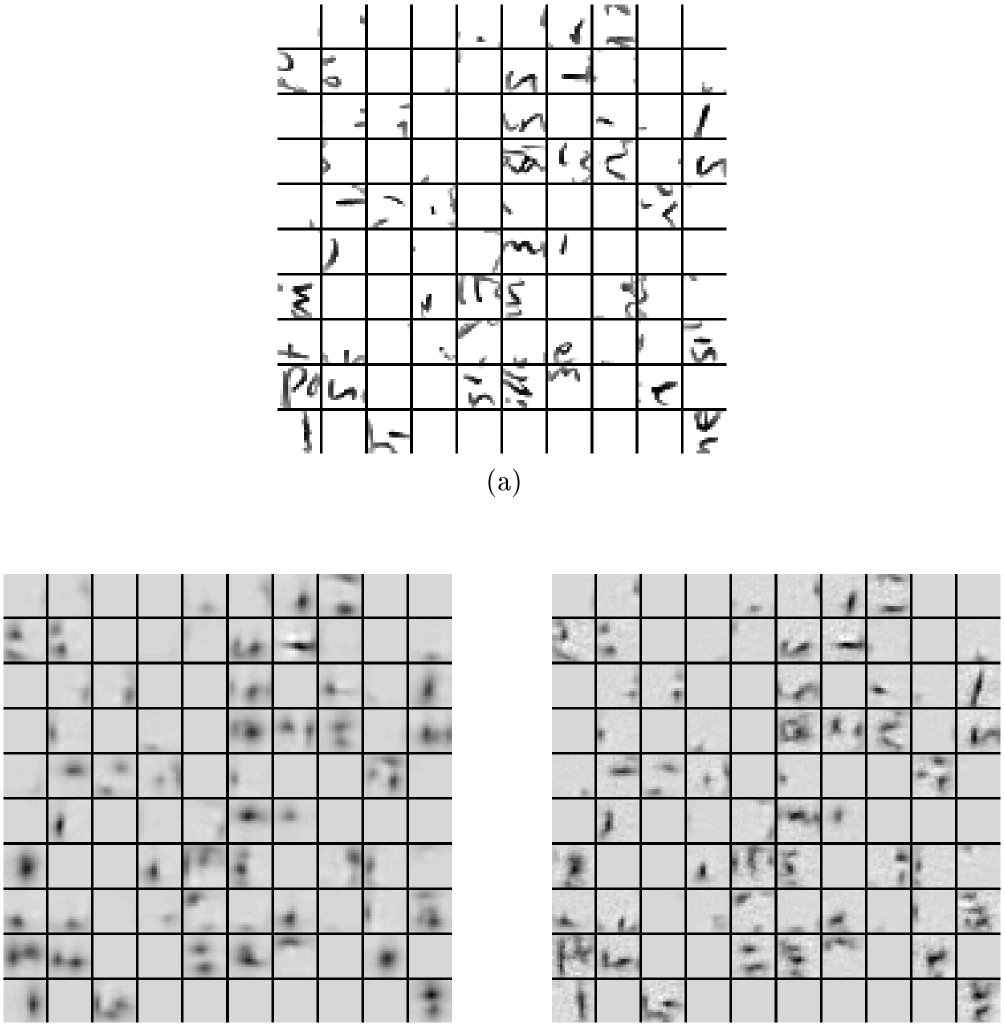


Figure 4.13: A 2-dimensional map with 400 neurons has learned samples taken from hand-written text. The samples have been weighted with a Gaussian function (see figure 4.12 (a)).



(a) (b) (c)

Figure 4.14: 100 samples (a) have been linerly reconstructed from the outputs of a map with 100 neurons (b) (the map in figure 4.12) and 400 neurons (c) (the map in figure 4.13). The reconstructions give an idea about the amount of information in the outputs.

Chapter 5

Discussion

Sparse coding is a biologically motivated way to represent information. It combines advantages of local and dense coding while avoiding most of their drawbacks. In this work a computationally efficient algorithm for finding sparse codes has been developed. It is able to mimic some aspects of the sensory processing in the brain, which emerge from the competition between neurons, although the exact mechanisms underlying the competition are apparently different.

Simulations with artificial and natural data show that the algorithm is able to find a meaningful representation to its inputs. It can be argued that the algorithm is able to find objects, because the similarities between our intuitive view about objects and the representations, which the algorithm has found, are quite striking. Objects (and also features, concepts or anything that deserves a name) can be described as collections of highly correlated properties (Barlow, 1985). For instance, the properties 'furry', 'has tail', 'moves', 'animal', 'barks', etc. are highly correlated, that is, the combination of these properties is much more frequent than it would be if they were independent. Therefore the collection of these properties deserves a name. It seems that the algorithm developed in this work is able to learn to group this kind of correlated features, and moreover is able to do so even when there are always several of these objects present in the data simultaneously.

Unlike in the case of a linear network, it may be useful to consider a hierarchical arrangement of several layers of sparse coding networks. Each layer can group features from the preceding layers into more abstract features, which can in turn be used in the following layers. This way the system could gradually find more and more abstract features to represent the inputs. This kind of ability to make abstractions is considered to be one of the fundamental properties of human intelligence.

It is very tempting to think that the similarities between the representations learned by the algorithm and our notion about objects is due to the underlying similarities in the information processing principles between the algorithm and the brain. Even if this was not true, the algorithm is well suited for automatically extracting features in engineering applications. This is due to the many advantageous properties of sparse codes and the efficiency of the algorithm proposed in this work. The computational complexity of the algorithm is of order NM, where N is the number of neurons and M is the number of inputs, as opposed to previous algorithms, where the complexity has typically been quadratically dependent on the number of neurons. Maybe the most prominent applications for the algorithm would be in preprocessing. The simulations with digitised samples of handwritten text show that the algorithm is able to find features even if the data has not been preprocessed in any way.

A very important future line of study would be the extension of the sparse coding algorithm into time domain. Currently the algorithm can group features that occur simultaneously, but it might be possible to group features that occur close in time. The problem to be solved would be the implementation of the competition mechanism: each neuron should be able to compete with neurons that have been previously active in addition to other neurons that are simultaneously active, because previous activities might convey information about events that are still going on. This kind of extended algorithm would be able to find dynamic features, which would be very useful in many applications. This algorithm, too, can be used to find dynamic features, if some dynamic preprocessing is applied to the data. It would be better, however, if the algorithm could directly take into account the dynamic structure of the input.

Bibliography

Abut, H., editor (1990). *Vector Quantization.* IEEE Press, New York.

Atkinson, K. E. (1989). *An Introduction to Numerical Analysis.* John Wiley & Sons, New York, 2^{nd} edition.

Attneave, F. (1954). Informational aspects of visual perception. *Psychol. Rev.*, 61:183–193.

Baddeley, R. (1996). Searching for filters with "interesting" output distributions: an uninteresting direction to explore? *Network.* In press.

Barlow, H. B. (1960). The coding of sensory messages. In Thorpe, W. H. and Zangwill, O. L., editors, *Current Problems in Animal Behaviour*, pages 331–360. Cambridge University Press, Cambridge.

Barlow, H. B. (1985). Cerebral cortex as model builder. In Rose, D. and Dobson, V. G., editors, *Models of the visual cortex*, pages 37–46. Wiley, Chichester.

Dayan, P. and Zemel, R. S. (1995). Competition and multiple cause models. *Neural Computation*, 7(3):565–579.

Field, D. J. (1994). What is the goal of sensory coding? *Neural Computation*, 6(4):559–601.

Földiák, P. (1990). Forming sparse representations by local anti-Hebbian learning. *Biological Cybernetics*, 64:165–170.

Földiák, P. and Young, M. P. (1995). Sparse coding in the primate cortex. In Arbib, M. A., editor, *The Handbook of Brain Theory and Neural Networks*, pages 895–898. The MIT Press, Cambridge, Massachusetts.

Haykin, S. (1994). *Neural Networks.* Macmillan College Publishing Company, New York.

Hinton, G. E., McClelland, J. L., and Rumelhart, D. E. (1986). Distributed representations. In Rumelhart, D. E., McClelland, J. L., and the PDP Research Group, editors, *Parallel Distributed Processing: Explorations in the Microstructure of*

Cognition, Volume 1: Foundations, pages 77–109. The MIT Press, Cambridge, Massachusetts.

Hinton, G. E. and Zemel, R. S. (1994). Autoencoders, minimum description length and Helmholtz free energy. In Cowan, J. D., Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems 6*, pages 3–10. Morgan Kaufmann Publishers, San Francisco.

Hodges, J., Salmon, D., and Butters, N. (1992). Semantic memory impairments in Alzheimer's disease. *Neuropsychologia*, 30:301–314.

Kohonen, T. (1995). *Self-Organizing Maps*. Springer-Verlag, Berlin-Heidelberg-New York.

Meunier, C. and Nadal, J.-P. (1995). Sparsely coded neural networks. In Arbib, M. A., editor, *The Handbook of Brain Theory and Neural Networks*, pages 899–901. The MIT Press, Cambridge, Massachusetts.

Newsome, W. T., Britten, K. H., and Movshon, J. A. (1989). Neuronal correlates of a perceptual decision. *Nature*, 341:52–54.

Oja, E. (1995). Principal component analysis. In Arbib, M. A., editor, *The Handbook of Brain Theory and Neural Networks*, pages 753–756. The MIT Press, Cambridge, Massachusetts.

Olshausen, B. A. and Field, D. J. (1995). Sparse coding of natural images produces localized, oriented, bandpass receptive fields. Technical Report CCN-110-95, Department of Psychology, Cornell University, Ithaca, New York 14853.

Ritter, H. and Schulten, K. (1986). On the stationary state of Kohonen's self-organizing sensory mapping. *Biological Cybernetics*, 54:99–106.

Rumelhart, D. E. and Zipser, D. (1986). Feature discovery by competitive learning. In Rumelhart, D. E., McClelland, J. L., and the PDP Research Group, editors, *Parallel Distributed Processing: Explorations in the Microstructure of cognition, Volume 1: Foundations*, pages 151–193. The MIT Press, Cambridge, Massachusetts.

Salzman, C. D. and Newsome, W. T. (1994). Neural mechanisms for forming a perceptual decision. *Science*, 264:231–237.

Saund, E. (1994). Unsupervised learning of mixtures of multiple causes in binary data. In Cowan, J. D., Tesauro, G., and Alspector, J., editors, *Advances in Neural Information Processing Systems 6*, pages 27–31. Morgan Kaufmann Publishers, San Francisco.

Saund, E. (1995). A multiple cause mixture model for unsupervised learning. *Neural Computation*, 7(1):51–71.

Schmidhuber, J. (1992). Learning factorial codes by predictability minimization. *Neural Computation*, 4(6):863–879.

Sirosh, J. (1995). *A Self-Organizing Neural Network Model Of The Primary Visual Cortex.* PhD thesis, University of Texas at Austin.

Sirosh, J. and Miikkulainen, R. (1994). Cooperative self-organization of afferent and lateral connections in cortical maps. *Biological Cybernetics*, 71(1):65–78.

Thorpe, S. (1995). Localized versus distributed representations. In Arbib, M. A., editor, *The Handbook of Brain Theory and Neural Networks*, pages 549–552. The MIT Press, Cambridge, Massachusetts.

Watanabe, S. (1981). Pattern recognition as a quest for minimum entropy. *Pattern Recognition*, 13:381–387.

Zemel, R. S. (1993). *A minimum description length framework for unsupervised learning.* PhD thesis, University of Toronto, Canada.

Appendix A

Derivations

A.1 Derivation of the equation 3.2

We first prove that equation 3.2:

$$y = \max(0, \boldsymbol{w}^T \boldsymbol{x}), \tag{A.1}$$

is the solution to the system of the following equations:

$$\hat{\boldsymbol{x}} = \boldsymbol{g}(y) = y\boldsymbol{w} \tag{A.2}$$

$$||\boldsymbol{w}|| = 1 \tag{A.3}$$

$$\mathcal{E}(\boldsymbol{x}, \hat{\boldsymbol{x}}) = ||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2 \tag{A.4}$$

$$y = \arg\min_z \mathcal{E}(\boldsymbol{x}, \boldsymbol{g}(z)) \tag{A.5}$$

$$y \geq 0 \tag{A.6}$$

We assume here that the weight vector $\boldsymbol{w}$ is a column vector with the same dimensionality as $\boldsymbol{x}$.

At first we combine equations A.2, A.4 and A.5. They yield

$$y = \arg\min_z ||\boldsymbol{x} - z\boldsymbol{w}||^2.$$

We can further manipulate the norm appearing on the right hand side of the equation.

$$||\boldsymbol{x} - z\boldsymbol{w}||^2 = (\boldsymbol{x} - z\boldsymbol{w})^T(\boldsymbol{x} - z\boldsymbol{w}) =$$

$$\boldsymbol{x}^T\boldsymbol{x} - 2z\boldsymbol{w}^T\boldsymbol{x} + z^2\boldsymbol{w}^T\boldsymbol{w} = ||\boldsymbol{x}||^2 - 2z\boldsymbol{w}^T\boldsymbol{x} + z^2$$

Here we have used the fact that $\boldsymbol{w}^T\boldsymbol{w} = ||\boldsymbol{w}||^2 = 1$. The term $||\boldsymbol{x}||^2$ does not depend on z, and therefore it can be left out without affecting the minimisation. We can now rewrite the system of equations:

$$y = \arg\min_z (z^2 - 2z\boldsymbol{w}^T\boldsymbol{x})$$

$$y \geq 0$$

Without the constraint $y \geq 0$ the solution would be $y = \boldsymbol{w}^T\boldsymbol{x}$. It is evident that equation A.1 gives the solution when we take the constraint into consideration.

A.2 Derivation of the equation 3.4

Next we prove that equation 3.4:

$$y_1 = \begin{cases} \max\left[0, \min\left(p_1, \frac{p_1 - cp_2}{1-c^2}\right)\right] & p_1, p_2, c > 0 \\ p_1 & p_1 > 0 \text{ and } (p_2 \leq 0 \text{ or } c \leq 0) \\ 0 & p_1 \leq 0 \end{cases}, \tag{A.7}$$

where $p_i = \boldsymbol{w}_i^T\boldsymbol{x}$ and $c = \boldsymbol{w}_1^T\boldsymbol{w}_2$, is the solution to the system of the following equations:

$$\boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \tag{A.8}$$

$$W = \begin{bmatrix} \boldsymbol{w}_1 & \boldsymbol{w}_2 \end{bmatrix} \tag{A.9}$$

$$\hat{\boldsymbol{x}} = \boldsymbol{g}(\boldsymbol{y}) = \sum_i y_i\boldsymbol{w}_i = W\boldsymbol{y} \tag{A.10}$$

$$||\boldsymbol{w}_i|| = 1 \tag{A.11}$$

$$\mathcal{E}(\boldsymbol{x}, \hat{\boldsymbol{x}}) = ||\boldsymbol{x} - \hat{\boldsymbol{x}}||^2 \tag{A.12}$$

$$\boldsymbol{y} = \arg\min_{\boldsymbol{y}} \mathcal{E}(\boldsymbol{x}, \boldsymbol{g}(\boldsymbol{z})) \tag{A.13}$$

$$y_i \geq 0 \tag{A.14}$$

$$y_i \leq \max(0, \boldsymbol{w}_i^T \boldsymbol{x}) \tag{A.15}$$

We assume here that the weight vectors $\boldsymbol{w}_i$ are column vectors with the same dimensionality as $\boldsymbol{x}$.

We know that the equations do not have a unique solution if the weight vectors are identical. In that case the correlation between the weight vectors would equal one. We assume that this degenerate case can be omitted, and we always have $c \neq 1$.

The derivation is very similar to the previous one. At first we combine equations A.10, A.12 and A.13, which yields

$$\boldsymbol{y} = \arg\min_{\boldsymbol{z}} ||\boldsymbol{x} - W\boldsymbol{z}||^2.$$

We can further manipulate the norm appearing on the right hand side of the equation. We denote the norm by $N(\boldsymbol{z})$.

$$N(\boldsymbol{z}) = ||\boldsymbol{x} - W\boldsymbol{z}||^2 = (\boldsymbol{x} - W\boldsymbol{z})^T(\boldsymbol{x} - W\boldsymbol{z}) = \boldsymbol{x}^T\boldsymbol{x} - 2\boldsymbol{z}^T W^T \boldsymbol{x} + \boldsymbol{z}^T W^T W \boldsymbol{z}$$

Using the notations $p_i = \boldsymbol{w}_i^T \boldsymbol{x}$, $\boldsymbol{p} = [p_1, p_2]^T$ and $c = \boldsymbol{w}_1^T \boldsymbol{w}_2$, and substituting $||\boldsymbol{w}_i|| = 1$ we can write

$$W^T\boldsymbol{x} = \begin{bmatrix} \boldsymbol{w}_1^T \boldsymbol{x} \\ \boldsymbol{w}_2^T \boldsymbol{x} \end{bmatrix} = \boldsymbol{p}$$

$$W^T W = \begin{bmatrix} \boldsymbol{w}_1^T \boldsymbol{w}_1 & \boldsymbol{w}_1^T \boldsymbol{w}_2 \\ \boldsymbol{w}_2^T \boldsymbol{w}_1 & \boldsymbol{w}_2^T \boldsymbol{w}_2 \end{bmatrix} = \begin{bmatrix} 1 & c \\ c & 1 \end{bmatrix}.$$

Let us denote the square matrix $W^T W$ by S. We can now write

$$\boldsymbol{y} = \arg\min_{\boldsymbol{z}} N(\boldsymbol{z}) = \arg\min_{\boldsymbol{z}} (\boldsymbol{x}^T\boldsymbol{x} - 2\boldsymbol{z}^T\boldsymbol{p} + \boldsymbol{z}^T S \boldsymbol{z}).$$

Note that because S is positive definite (we assumed $c \neq 1$), $N(\boldsymbol{z})$ is a parabolic function.

We first find the solution without the constraints A.14 and A.15. If $\boldsymbol{y}$ is the solution to the equation, then the gradient of $N(\boldsymbol{z})$ at point $\boldsymbol{y}$ has to be a zero vector $\mathbf{0}$. The gradient of N is

$$\nabla_{\boldsymbol{z}} N = -2\boldsymbol{p} + 2S\boldsymbol{z},$$

and therefore the solution $\boldsymbol{y}$ has to satisfy

$$-2\boldsymbol{p} + 2S\boldsymbol{y} = \mathbf{0} \Leftrightarrow \boldsymbol{y} = S^{-1}\boldsymbol{p}.$$

The computation of S^{-1} yields

$$S^{-1} = \frac{1}{1 - c^2} \begin{bmatrix} 1 & -c \\ -c & 1 \end{bmatrix},$$

and thus the solution without the constraints is

$$\boldsymbol{y} = \frac{1}{1-c^2}\begin{bmatrix} p_1 - cp_2 \\ p_2 - cp_1 \end{bmatrix}. \tag{A.16}$$

Next we take into consideration the constraints A.14 and A.15. If either of the p_i is less than or equal to zero, then $y_i = 0$ and the solution to the other output reduces to the case of one neuron, which has already been solved. We can thus assume that both $p_i > 0$. The constraints bound a rectangular area of allowed outputs. If the unconstrained solution given by equation A.16 lies in that area, it is also the solution to the constrained minimisation. Those cases, where the unconstrained solution does not lie in the allowed area, are left to examine. Due to the symmetry, the solutions to y_1 and y_2 are identical except for the interchanged indices. We can therefore consider only the solution to y_1. Let us see what happens if $c \leq 0$. It is evident that the solutions to both of the outputs y_i in equation A.16 exceed the boundaries:

$$y_1 = \frac{p_1 - cp_2}{1-c^2} \geq p_1 - cp_2 \geq p_1$$

Because $N(\boldsymbol{z})$ is parabolic, the best constrained solution lies in the corner of the allowed area: $y_1 = p_1$ and $y_2 = p_2$.

Now only those cases are left where p_1, p_2, and $c > 0$. Let us consider what happens if equation A.16 gives a solution $y_1 < 0$. We now that $\boldsymbol{S}\boldsymbol{y} = \boldsymbol{p} \Leftarrow cy_1 + y_2 = p_2$, and therefore $y_1 < 0 \Leftrightarrow y_2 > p_2$. We see that if the unconstrained solution to one of the outputs is under the bound, then the other is over, and vice versa. Again the constrained solution is found in the corner of the allowed area. This time one of the outputs is zero and the other is p_i. The solution to the case $p_1, p_2, c > 0$ can thus be formulated

$$y_1 = \max\left[0, \min\left(p_1, \frac{p_1 - cp_2}{1-c^2}\right)\right].$$

Collecting together all the different cases yields the equation 3.4. This concludes the proof.