# MODELLING COGNITIVE REPRESENTATIONS

a Master Thesis

by Morten Arngren, s030192

on March 22nd, 2007

# Preface

This thesis constitutes part of the requirement for obtaining the Master of Science degree at the Technical University of Denmark. The work was carried out in the period September 1st 2006 to March 22nd 2007 at the *Intelligent Signal Processing* group at *Informatics and Mathematical Modelling* institute and at the penthouse in Badstuestræde.

Before commencing the presentation of the thesis, I would like to express my gratitude to my supervisor throughout the project Associate Professor Ole Winther for keeping inspiring and guiding me through the thesis.

Kgs. Lyngby, March 22nd, 2007

Morten Arngren, s030192

# Abstract

This thesis analyzes the modelling of visual cognitive representations based on extracting cognitive components from the MNIST dataset of handwritten digits using simple unsupervised linear and non-linear matrix factorizations both non-negative and unconstrained based on gradient descent learning.

We introduce two different classes of generative models for modelling the cognitive data: *Mixture Models* and *Deep Network Models*.

Mixture models based on K-Means, Guassian and factor analyzer kernel functions are presented as simple generative models in a general framework. From simulations we analyze the generative properties of these models and show how they render insufficient to proper model the complex distribution of the visual cognitive data.

Motivated by the introduction of deep belief nets by Hinton et al. [12] we propose a simpler generative deep network model based on cognitive components. A theoretical framework is presented as individual modules for building a generative hierarchical network model.

We analyze the performance in terms of classification and generation of MNIST digits and show how our simplifications compared to Hinton et al. [12] leads to degraded performance. In this respect we outline the differences and conjecture obvious improvements.

**Keywords** : *Cognitive Component Analysis, Generative Models, Matrix Factorization, Mixture Models, EM-algorithm, Deep Network models*

# Contents

# Nomenclature

Throughout the thesis the following notation is used.

| | | |
|---|---|---|
| $x$ | : | Scalar |
| $\mathbf{x}$ | : | Column vector |
| $\mathbf{x}_{i,n}$ | : | The n'th datasample at subcoordinate i. |
| $\mathbf{X}$ | : | Matrix |
| | | |
| $\alpha$ | : | Regulation term for the sources $\mathbf{s}$. |
| $\beta$ | : | Regulation term for the codebook matrix $\mathbf{A}$. |
| $\gamma$ | : | Regulation term for class. weight matrix $\mathbf{W}$. |
| $\sigma^2$ | : | Balance Parameter. |
| $d$ | : | Dimension of features / number of latent variables. |
| $h$ | : | Hidden or latent variable. |
| $m$ | : | Index for elements in $\mathbf{x}$. |
| $n$ | : | Index for samples. |
| $r$ | : | Index for features, column vectors in $\mathbf{A}$. |
| $y$ | : | Class conditional probability. |
| $M$ | : | Dimension of data. |
| $N$ | : | Number of datasamples. |
| $\mathbf{s}$ | : | Latent source or codevector. |
| $\mathbf{t}$ | : | Target vector, binary. |
| $\mathbf{x}$ | : | Observed sample. |
| $\mathbf{A}$ | : | Codebook with column vectors as features. |
| $\mathbf{H}$ | : | Hidden variables as vectors. |
| $\mathbf{S}$ | : | Source matrix. |
| $\mathbf{W}$ | : | Weights used for classification. |
| | | |
| $\mathcal{E}\{\cdot\}$ | : | Expectation, statistical average. |
| $p(\mathbf{x})$ | : | Probability distribution of $\mathbf{x}$. |
| $\mathcal{N}(\mu, \mathbf{\Sigma})$ | : | Gaussian distribution with mean $\mu$ and covariance matrix $\mathbf{\Sigma}$. |
| $\mathcal{H}(x)$ | : | Entropy function of x. |
| $\mathcal{L}_{inc}(\cdot)$ | : | Incomplete Log-Likelihood. |
| $\mathcal{L}_{c}(\cdot)$ | : | Complete Log-Likelihood. |
| $\mathrm{KL}\big(p(x) \mid\mid p(y)\big)$ | : | Kullback-Leibler divergence. |

# 1. Introduction

In the recent 50+ years the development of the computer has given us a tool with immense power. Problems, which before seemed impossible or at least complex can now be solved within fractions of a second. Simple mathematical tasks are solved easily by a computer superior to the human brain. Just compare the time to give $\frac{1}{6}$ in decimals between a human (with a pencil, paper and a few minutes) and a computer. Most people might not even give an answer, whereas a computer is virtually infinitely fast. But if the task is to recognize a song from the radio or perhaps write zipcode digits on a letter, the human brain has superior skills and is hardly challenged. A computer on the other hand requires advanced algorithms and may need considerable time to perform the task if even possible.

Throughout the history and literature we have always thrived to investigate and analyze the ever complex working of the brain in every aspect ranging from simple insect behavior to the cognition of the human mind. The human brain and its intelligence has always been considered as a superior tool for information processing and being able to describe its mysteries may serve as a foundation for developing complex computer based algorithms in every aspect. We have indeed come a long way in every field such as psychology, medicine, machine learning etc., but since every revelation always calls for new quests the 'finish-line' will always be ahead of us.

The processing of the brain can be analyzed in numerous ways and angles and in this thesis we will focus on the field of modelling cognitive visual data. This is however not a new research field, plenty of literature and numerous articles have described the visual processing system, refer to [1] [28] [13] [14] as a short reference. Common of these articles is that their description only includes the relatively simple initial processing layers (LGN and V1) and merely speculates on the further and vastly more complex processing of visual stimuli.

This thesis will try to enlighten the modelling of visual data by introducing *Cognitive Component Analysis* (COCA) as a tool for analyzing and modelling cognitive learning. A mathematical framework will be presented, implemented in MATLAB and finally evaluated through simulations on small images of handwritten digits (the MNIST dataset). A short quantitative introduction to the visual cortex is described next to serve as a small background.

## 1.1 Visual Cortex

Understanding the sensory neurons of the visual system in the brain has received much attention as a primary research area in the last many decades. The main approach is attempting to model the sensory processing with linear modelling based on the statistical properties of the environment. Here we will try to give a very short overview of the main issues for the neurological visual processing system based on [1] [13] [14] [15] [28].

The visual cortex is part of the neural visual center of the brain and occupies about one third of the surface of the cerebral cortex in humans, illustrated in figure 1.1 [1].

The signal path in the brain can be described very shortly from a macro-perspective point of view. Visual stimuli from the natural world enters the eye and is projected on the retina (photo sensitive backside of the eye) and from here the Lateral Geniculate Nucleus (LGN) receives the visual information. The LGN is considered the primary processor of visual information and together with the retina, they are thought to eliminate the correlation and remove redundancy in the visual stimuli as stated by Hoyer & Hyvärinen [14] and Simoncelli & Olshausen [28]. The image data then still contains obvious structures like lines, edges contours etc. after decorrelating and thus in terms of efficient coding in the neurons much work is still to be done [28]. Redundancy reduction techniques in terms of *Principal Component Analysis* (PCA) are shortly presented in appendix A.1.

From here the signal is passed to the visual cortex, which very simplified can be split up in a *ventral* and *dorsal* stream with numerous hierarchical levels of increasingly complex processing. The ventral stream is believed to be responsible for object recognition (also called the 'what' stream), where the dorsal stream is representing movement (also called the 'where' stream). The depth of the hierarchical layers are thought to extend to 30 levels or more. The initial layers of both streams are denoted V1-V5 and are only the tip of the iceberg in visual

[1]Image taken from http://www.nmr.mgh.harvard.edu/∼rhoge/HST583/doc/HST583-Lab1.html

cognition. We will venture to try to give a short and very simplified description of this complex processing of initial these layers. The V1 area (primary visual cortex) is the first and most simple of the levels. Due to the pioneering work of by Hubel and Wiesel [15] based on experiments with macaque monkeys exposed to visual input much is known about the processing of visual input in V1.

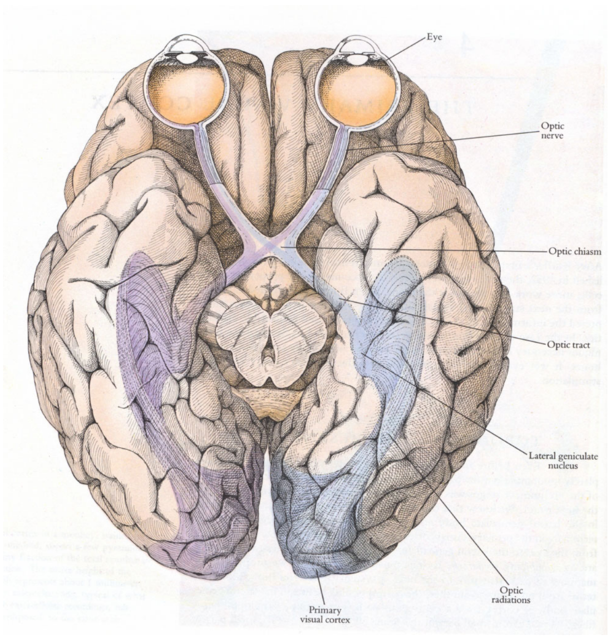

*Figure 1.1:* Physical outline of the visual cortex.

The neurons in V1 can be split up in simple, complex and hypercomplex cells, where the simple cells are thought to perform some linear filtering tuned to simple edge detection of the visual input. The spatial characteristics of these elongated filters (receptive fields) can be described as having oriented and bandpass features [13] [28] [5]. It is exactly this Gabor-like structure in the simple cells that gives the exhibitory and inhibitory functionality suitable for edge detection. The complex and hypercomplex cells in V1 perform similar filtering, but are without clear excitatory or inhibitory zones (i.e. are not tuned for edge detection) and are more sensitive to complex non-local structures, e.g. two edges at right angles to each other in a yet larger region of the visual field. This suggest that the complex and hypercomplex cells receive their input from lower level simple neurons in V1 [15].

It has been argued by Simoncelli & Olshausen [28] that neural responses must be statistically independent in order to avoid duplication of information in the neurons and hence for the coding of information to be efficient. Perhaps for the purpose of accurate spatial encoding, neurons in V1 have the smallest receptive field size of any visual cortex regions. By applying the linear *Independent Component Analysis* (ICA) model with independence as the constraint to natural images leads to features similar to those found in receptive fields of simple cells in V1 [1] [13] [28]. Similar results has also been found using *Linear Sparse Coding* (LSC), where image data is decomposed into a linear model under the constraint that the components must to be as sparse as possible [13] [25]. Additional cells in V1 are suspected to extract color information inside blobs (columns of cells with circular receptive fields) and further to hold binocular cells responsible for stereopsis or depth information.

Further neural processing of the sensory input based on the architecture of the visual cortex suggests a hierarchical organization, where the neurons becomes increasingly selective to more complex image structures [28]. The second region denoted V2 is the first level within the visual association area. The responses of many V2 neurons are also modulated by more complex properties, such as the orientation of illusory contours and whether the stimulus is part of the figure or the ground. Properties of cells in the next layer V3 offer few clues as to its function. Most are selective for orientation and many are also tuned to motion and to depth, but relatively few are color sensitive. The area in the deeper layer V4 contains many cells that are color selective, indicating a role in color analysis. Still cells are also found with complex spatial and orientation tuning, suggesting that the area is also important for spatial vision.

The deep level of V5 or middle temporal (MT) is the most prominent part of the cerebral cortex specialized for analyzing visual motion and is hence part of the dorsal stream. Most cells in V5 are tuned to motion, and the area can be divided into direction and 'axis of motion' columns. The receptive fields in V5 are in addition much larger than those in V1 making these cells inappropriate for object recognition as they would give a blurry image.

Further processing at least in the ventral stream suggest neurons exhibit properties that are important for object recognition. At the highest levels in this pathway, neurons in the inferior temporal cortex respond preferentially to faces. Furthermore, lesions to these areas produce prosopagnosia, a deficit in face recognition.

Our description of the visual cortex reveals a sequential structure from V1 to V5. This is somewhat simplified as the actual architecture is by far more complex with feedback and interlayer connections. To summarize simple, complex and hypercomplex cells can work together to decompose the outlines of a visual image into short segments, the basis of simple and complex object recognition. Higher level processing in V2-V5 are in charge of increasingly complex structures and movement in the visual input [2]. This multilayered hierarchical architecture of the visual cortex is the very foundation for deep networks and serves as motivation to one of models we build and analyze in this thesis.

The processing of the visual cortex for all levels suggests that localized oriented filters could be an optimal extraction of features in modelling visual data. This approach leads to the motivation of analyzing cognitive models in terms of cognitive components.

## 1.2 Cognition and Cognitive Components

The concept of *cognition* becomes evident in comparing or even describing the differences between the human brain and a computer. The human perceptional systems can model complex multi-agent scenery based on a broad spectrum of cues for analyzing perceptual input and separate individual signal producing agents, such as speakers, gestures, affections etc. Computers are in contrast superior in solving problems based on strict rules as opposed to cognitive input.

From the online version of *Britannica* cognition is defined as the "‘act or process of knowing"’ and continues

> *Cognition includes every mental process that may be described as an experience of knowing (including perceiving, recognizing, conceiving, and reasoning), as distinguished from an experience of feeling or of willing.*

The process of grouping events or objects into classes or categories is fundamental to human cognition. In machine learning the grouping of data based on labeled example is a widespread technique known as *supervised* learning [4]. In contrast clustering data without a priori providing a set of labeled examples is an *unsupervised* learning problem, where we use a set of general statistical rules to group the objects. For many real world data sets the labeled structure inferred by unsupervised learning closely coincide with labels obtained from human manual classification, i.e. labels derived from human cognition [8].

It has long been assumed that neurons adapted at evolutionary development and behavioral timescale to the signal to which they are exposed [28]. This fact suggests that the receptive fields in the visual cortex has been formed from unsupervised learning and coincide the our definition of cognitive components. We therefore base our entire analysis on unsupervised learning as the fundament.

One of the more recent research fields is the discipline of *Cognitive Component Analysis* (*COCA*). The objective in COCA is to perform unsupervised grouping of data such that the ensuing group-structure is well-aligned with that resulting from human cognitive activity [8]. In other words cognitive components can be defined as segments, which gives ’perceptual meaning’ to humans. As an example figure 1.2 illustrates how the image of a 2 digit can be decomposed into subparts which are consistent with the intuitive segmentation performed by a human and can hence be characterized as cognitive components according to our definition.

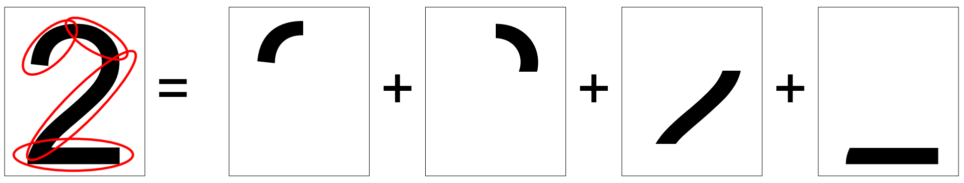

*Figure 1.2:* Split-up of the 2 digit into cognitive components.

Modelling cognitive components naturally leads to using generative models as the cognitive components then form the basic generative weights or features to allow efficient representation. In using a generative model we capture information used to model the complete structure of the observed data as opposed to a strictly discriminative model, where only classification information is extracted. For class. purposes a generative model holding efficient representing of the data has shown to have superior performance over discriminative models

[2]For a visual demo of the visual cortex, visit http://www.physpharm.fmd.uwo.ca/undergrad/medsweb/L2V123/M2-V123.swf.

[11] [12]. In addition the linear filtering in the visual cortex serves as extra motivation for using generative models for modelling cognitive data.

If we denote an observed $M$ dimensional random variable $\mathbf{x} = \{x_m\}_{m=1}^{M}$ a linear generative model is then as the name implies defined as a model, which can be used to generate new random data $\mathbf{x}$ with the same statistical properties as the observed $\mathbf{x}$. This means we decompose the observed data $\mathbf{x}$ into

$$\mathbf{x} = \mathbf{A}\mathbf{s} \tag{1.1}$$

where the matrix $\mathbf{A} = \{\mathbf{a}_r\}_{r=1}^{d}$ is denoted the codebook holding generative features of $\mathbf{x}$ and the $d$ dimensional vector $\mathbf{s}$ is the corresponding latent source or encoding vector, which contains the weights of each feature $\mathbf{a}_r$ used to form $\mathbf{x}$. The extraction of features through matrix decomposition in (1.1) means we can express the observed $\mathbf{x}$ from a compressed coded vector $\mathbf{s}$ provided $d < M$ leading to potentially less complex modelling of the cognitive data as opposed to modelling $\mathbf{x}$ directly.

In the context of cognitive analysis we aim to decompose $\mathbf{x}$ such that the codebook $\mathbf{A}$ holds cognitive components $\mathbf{a}_r$ as features. This means the method used to decompose $\mathbf{x}$ must be able to extract cognitive components and depending on the data type and context different methods can be used. If independence is an important constrain to ensure efficient coding *Independent Component Analysis* (ICA) can be used to find cognitive components. However the independence constrain in itself does not guarantee cognitive components, e.g. in the context of phonemes it has been shown that ICA can extract cognitive features [6], whereas for color image data ICA extracts localized Gabor-like filters [1] [14], which are hard to interpret visually and cannot be considered as cognitive components. For image data, which only contains non-negative pixels *Non-Negative Matrix Factorization* (NMF) has shown to give features, which in simple B/W cases are similar to those in figure 1.2 and can be considered cognitive [13].

As extraction of cognitive components are performed subjectively, they may not necessarily be unique. The exhibitory part of the structure-selective filters believed to be used in the processing in V1 of the visual cortex as discussed earlier can for instance from a subjective point of view be considered as cognitive components. For the complex and hypercomplex neurons in the primary visual cortex the inhibitory effect is believed to have less impact, i.e. the negative or subtractive region is not essential. In addition the firing rates from neurons cannot be negative [13] and it therefore seems justified to base our models on such non-negative component features through NMF decomposition.

For true modelling of the receptive fields of the simple cells in V1 it is essential to obtain sparse oriented Gabor-like features and for that purpose both ICA and an extended version of NMF has shown to give features similar to V1 simple neurons [1] [13] [14]. In addition Ranzato et al. has achieved similar sparse features without an independence constraint [25]. As a second approach we therefore base additional models on unconstrained regulated matrix factorizations without independence constraint to encourage sparse (simple cells) or abstract (hypercomplex cells) features with full dynamic range.

In our linear model (1.1) new data $\mathbf{x}$ can be generated from any random source vector $\mathbf{s}$. In order to generate a valid datapoint it is essential that a generated source maintains the correlation between the codebook elements in $\mathbf{A}$. In training a generative cognitive model it is crucial that the model captures this correlation and thus has the potential to generalize to valid data not covered by the training set. We can further identify these sources as representing *valid* cognitive data in dataspace. In contrast there exist sources $\mathbf{s}$ in dataspace, which are *invalid* in terms of cognitive data (this split-up will be more evident later). We can therefore label the dataspace into *valid* and an *invalid* areas and rephrase the learning task as to train the cognitive model to find the complex boundary between valid and invalid dataspace.

To summarize an efficient cognitive generative model will in this respect capture the underlying correlation or structure of the sources and generate new cognitive data. We propose two different models types as candidates for such efficient cognitive models, namely *The Mixture Models* and *Deep Network Models*.

## 1.3 The Models

In modelling cognitive data $\mathbf{x}$ we build a generative model based on unsupervised learning and focus on two different classes of models introduced below

### Mixture Models

From our linear representation of cognitive data $\mathbf{x}$ in (1.1) we can form a generative model by modelling the distribution of the sources $p(\mathbf{s})$. For this purpose we focus on the class of mixture models, where complex models can be assembled from mixtures of simpler kernel based functions with few parameters. The modelling strategy can be illustrated in a valid/invalid dataspace as in figure 1.3.

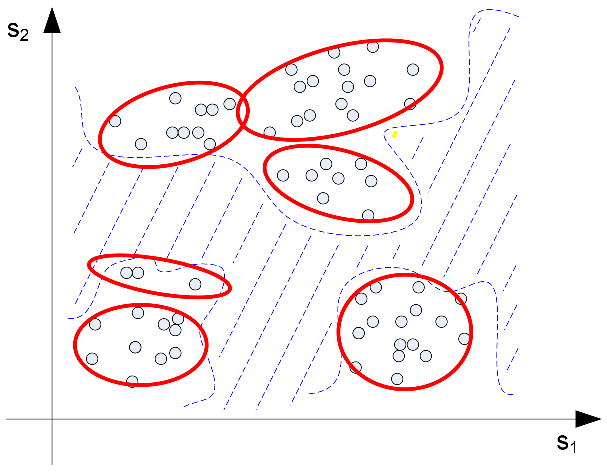

*Figure 1.3:* Gaussian based mixture models in a valid/invalid divided source space.

For probabilistic modelling a classic choice of a kernel function is the Gaussian probability distribution individually parameterized for the mixture components, denoted $K$. This approach is typically very powerful for modelling of 'localized' data, since $K(x, y)$ converges to a constant for increasing distance $\parallel x - y \parallel$. This however also sets a clear limitation when modelling 'high-varying' data, where a large amount of mixtures would be needed for sufficient modelling.

In addition for high dimensional data the Gaussian distribution risk suffering from the curse of dimensionality, where an exponentially increasing amount of samples are needed to define the parameters with equal precision for increasing dimensions. To accommodate for this limitation a kernel function based on the based on the *factor analyzer* function can be used instead, where a Gaussian distribution is modelled in sub-dimensional space.

With simple mixture models using unsupervised learning we aim to model the sources representing valid cognitive data capturing the cognitive correlation by the distribution $p(\mathbf{s})$.

### Deep Network Models

The class of Deep Network Models is based on multilayered non-linear generative functions, which does not suffer from 'localized' modelling as the mixture model does. Instead of building a deep network model from the decomposition in (1.1), we introduce a non-linear generalized factorization given by

$$\mathbf{x} = \mathbf{A}f(\mathbf{s}) \tag{1.2}$$

where $f(\mathbf{s})$ is a non-linear mapping function. This gives a far more adaptable model, where the linear factorization (1.1) now becomes a special case with $f(\mathbf{s}) = \mathbf{s}$. In such case the multilayered architecture could be collapsed into an equivalent 1-layer structure due to the linearity in (1.1) and thus the hierarchical structure is no longer appropriate.

Throughout the literature the 2-layer network has traditionally been the preferred structure as opposed to network with $L > 2$ layers. It has been shown empirically that multilayered networks are in fact considered worse in performance than networks with only 2-layers mainly due to the poor solutions often achieved from gradient-based optimizations from random initializations [2] [10].

Hinton et. al. has recently introduced a training algorithm for networks with multiple layers or *Deep Belief Nets*, where each layer is trained separately in an unsupervised greedy fashion leading to a crude approximation of an optimal solution. With such initialization of the network-layers the individual weights are afterwards fine-tuned to retrofit the parameters in each layer [12]. This modelling of cognitive data has resulted in representations lying in long ravines in subdimensional landscape leading to superior results in both generation and classification tasks [12]. The basic concepts in these deep belief nets are outlined on section 4.1 as a reference. Based on this approach deep network models has suddenly received wide attention as a promising research area for generative and classification models and in particular for the MNIST dataset classification [2] [12] [25].

The encouraging motivation for preferring multilayered architecture is the potential compact representation it may give. It may also give rise to a more cognitive interpretation of the individual layers. Upper layers learn and represent abstract concepts that explain the observation $\mathbf{x}$, where lower layers extract 'low-level' features

[2]. That is simpler concepts are initially learned and build on to learn more abstract concepts for lower layers [2]. This sequential processing is also believed occur in the brain as an potential optimal structure [28] and are well-aligned with the believed architecture of the visual cortex.

With the motivation from the deep net by Hinton et al. we propose to build a simpler generative deep network model, where each layer is trained separately unsupervised for modelling visual cognitive components. We will present a set of modules for both generation and classification of visual data and with different modelling properties, which can be used to assemble our deep network model. Our model is further simplified by not conducting any subsequent fine-tuning of model parameters after individual training of the layers as opposed to the deep belief net of Hinton et al.

In **section 2** a theoretical framework of mixture models is formed and analyzed as the basis for the first type of cognitive generative models. The mixture models are applied to the visual dataset MNIST in **section 3** and its performance evaluated in terms of generation of new data. Afterward deep network models are described theoretically in **section 4** as the second type of cognitive model. The network models are also subjected to the MNIST dataset in **section 5** and analyzed and evaluated in terms of generation and classification performance. Concluding remarks are made in the final **section 6**.

Part of the `MATLAB` code used for the simulations is listed in appendix B. A complete reference to the `MATLAB` code, dataset and results can be found on the associated data DVD.

# 2. Mixture Models

In the pursuit to generate valid data $\mathbf{x}$ based on our linear model $\mathbf{x} = \mathbf{A}\mathbf{s}$ in (1.1), we can express the probabilistically of the observed data $p(\mathbf{x})$ as

$$p(\mathbf{x}) = \int p(\mathbf{x}, \mathbf{s})\ \mathrm{d}\mathbf{s} = \int p(\mathbf{x}|\mathbf{s}) p(\mathbf{s})\ \mathrm{d}\mathbf{s} \tag{2.1}$$

where $p(\mathbf{s})$ is the unknown prior distribution of the sources and the conditional distribution $p(\mathbf{x}|\mathbf{s})$ models a Gaussian distribution representing our linear decomposition. This will be more evident later, refer to appendix A.6 for details.

In this section we present and outline the class of mixture models based on unsupervised learning. We previously highlighted the importance of maintaining correlation of the cognitive components in the sources $\mathbf{s}$ and for mixture models we aim to model this correlation and estimate valid sources from a density distribution of the sources $p(\mathbf{s})$ based on mixture models to approximate $p(\mathbf{s})$.

Mixture models is a wide area with many clustering algorithms in all degrees of complexity. In this analysis we will focus in primarily three simple mixture models

- *K-Means clustering*
- *Mixture of Gaussians (MoG)*
- *Mixture of Factor Analyzers (MFA)*

These mixture models will be presented in the following subsections. The paradigm in cognitive analysis is as mentioned unsupervised learning, where given data is modelled without prior knowledge of any relation or labeling. Obviously $\mathbf{x}$ can be decomposed in an infinitely number ways for our linear type of model and before we introduce the mixture models themselves, we describe a linear unsupervised non-negative matrix factorization for extracting features for our cognitive models.

## 2.1 Linear Non-negative Matrix Factorization

In certain real world applications, an observed dataset $\mathbf{X}$ can best be described as generated exclusively from adding components and not necessarily being independent. A decomposition into non-negative parts may in these cases give a more meaningful interpretation conceptually. Classic examples would be an RGB image, where subcolors (R, G or B) are added constructively to generate a color or it could be the wordcount from a document.

*Non-Negative Matrix Factorization* (NMF) seeks to factorize a matrix $\mathbf{X} = \{\mathbf{x}_n\}_{n=1}^{N}$, where $\mathbf{x}_n$ is an $M$ dimensional vector, into a low rank, sparse, non-negative linear approximation on the form

$$\mathbf{X} \approx \mathbf{A}\mathbf{S} \tag{2.2}$$

where the columns of $\mathbf{A} = \{\mathbf{a}_r\}_{r=1}^{d}$ are the $M$ dimensional basis vectors and the $d \times N$ matrix $\mathbf{S}$ holds the encoding, where the desired rank $d$ is chosen so that $(m + N)d < mN$. All matrices in (2.2) are under the constraint not to include any negative elements.

If we assume the individual datasamples $\mathbf{x}$ are generated from (1.1) with additive Gaussian noise $\epsilon$, we can expand the linear model into $\mathbf{x} = \mathbf{A}\mathbf{s} + \epsilon$, where the random variable $\epsilon$ is Gaussian distributed. This assumption is related to the class of linear models as shown in appendix A.6, where the maximum likelihood estimate of $p(\mathbf{x}|\mathbf{s})$ naturally leads to the least-squares cost function based on Euclidean distance. Thus to find the non-negative factorizations of $\mathbf{A}$ and $\mathbf{S}$, we seek to maximize the log-likelihood of $p(\mathbf{x}|\mathbf{s})$ and thus minimize a cost-function given by

$$
\begin{aligned}
E_{LS} &= \frac{1}{2\sigma^2} \parallel \mathbf{X} - \mathbf{AS} \parallel^2 && (2.3)\\
&= \frac{1}{2\sigma^2} \parallel (\mathbf{X} - \mathbf{AS})^T(\mathbf{X} - \mathbf{AS}) \parallel && (2.4)\\
&= \frac{1}{2\sigma^2} \parallel \mathbf{X}^T\mathbf{X} - \mathbf{X}^T\mathbf{AS} - (\mathbf{AS})^T\mathbf{X} + (\mathbf{AS})^T\mathbf{AS} \parallel && (2.5)\\
&= \frac{1}{2\sigma^2} \parallel \mathbf{X}^T\mathbf{X} - 2\mathbf{X}^T\mathbf{AS} + (\mathbf{AS})^T\mathbf{AS} \parallel && (2.6)
\end{aligned}
$$

We can further formulate the update of $\mathbf{A}$ and $\mathbf{S}$ as to minimize (2.6) by calculating the 1st order derivative of (2.3) and use a gradient descent type approach expressed as (only shown for the sources $\mathbf{S}$)

$$
\mathbf{S}_{r,n} \leftarrow \mathbf{S}_{r,n} - \Delta\frac{\partial E_{LS}}{\partial \mathbf{S}_{r,n}} = \mathbf{S}_{r,n} + \eta_{r,n}\Big[(\mathbf{A}^T\mathbf{X})_{r,n} - (\mathbf{A}^T\mathbf{AS})_{r,n}\Big] \tag{2.7}
$$

where we have assumed $\sigma^2 = 1$, since optimizing (2.7) wrt. $\mathbf{S}$ is independent of $\sigma^2$. To ensure a non-negative update $\mathbf{S}$ the gradient in (2.7) does not guarantee non-negativity as it can point to any orthant in multidimensional space. Instead we use a selective stepsize $\eta_{r,n}$ as opposed to a constant such that gradient directions leading to a negative orthant are suppressed. This can be achieved by setting $\eta_{r,n} = \frac{\mathbf{S}_{r,n}}{(\mathbf{A}^T\mathbf{AS})_{r,n}}$ and thus we can rewrite the update of $\mathbf{S}$ and similarly for $\mathbf{A}$ to :

$$
\mathbf{S}_{r,n} \leftarrow \mathbf{S}_{r,n}\frac{(\mathbf{A}^T\mathbf{X})_{r,n}}{(\mathbf{A}^T\mathbf{AS})_{r,n}} \qquad \wedge \qquad \mathbf{A}_{m,r} \leftarrow \mathbf{A}_{m,r}\frac{(\mathbf{XS}^T)_{m,r}}{(\mathbf{ASS}^T)_{m,r}} \tag{2.8}
$$

where the fraction indicates a component-wise division of the matrices. This forms an iterative coordinate-descent algorithm, where one parameter is updated, while the other is kept constant and vice versa. These update rules further guarantee convergence to at least a local optimum as proven in [18]. Note that if $\mathbf{X}$ is non-negative (i.e. contains only non-negative elements), the update of both $\mathbf{A}$ and $\mathbf{S}$ also remain non-negative. The column vectors of encoding matrix $\mathbf{A}$ are in addition scaled to unity length during the iterations, i.e. $|\mathbf{a}_r| = 1$ and sources $\mathbf{s}$ accordingly to keep values in reasonable dynamic range. This has no effect on the updates equation, as it is a linear scaling in both the nominator and denominator of in (2.8). Refer to [17] and [18] for a more detailed description on the update algorithm on NMF.

In the practical implementation the convergence speed is controlled dynamically by defining the step-size parameters $\eta$ and $\zeta$ as exponents for the multiplicative the update rules. Hence we rewrite eq. (2.8) into

$$
\mathbf{S}_{r,n} \leftarrow \mathbf{S}_{r,n}\left[\frac{(\mathbf{A}^T\mathbf{X})_{r,n}}{(\mathbf{A}^T\mathbf{AS})_{r,n}}\right]^{\eta} \qquad \wedge \qquad \mathbf{A}_{m,r} \leftarrow \mathbf{A}_{m,r}\left[\frac{(\mathbf{XS}^T)_{m,r}}{(\mathbf{ASS}^T)_{m,r}}\right]^{\zeta} \tag{2.9}
$$

During iterations the stepsizes $\eta$ and $\zeta$ are boosted or increased in case of a successful decrease of the cost function (2.3) and lowered if the step was too long resulting in an increase of the cost function $E_{LS}$. The practical implementation of NMF algorithm is provided as part of the SNMF2D toolbox from IMM [3] and is also available on the DVD. Refer to [26] for a more detailed description of the algorithm in the SNMF2D toolbox.

## 2.2 K-Means Clustering

One of the simplest classes of unsupervised learning is *Cluster Analysis*, which as the name implies based on clustering or data segmentation. The objective is to infer the structural properties of a given dataset $\mathbf{X}$ without 'labels' by grouping similar datapoints into subsets or clusters, such that those within a cluster are more related than those assigned to other clusters. In terms of building a generative model cluster analysis can be used to provide a simple tool for modelling either the codebook elements $\mathbf{A} = \{\mathbf{a}_r\}_{r=1}^{d}$ or the sources themselves $\mathbf{s}$.

[3]Available at http://www.imm.dtu.dk/pubdb/views/edoc_download.php/4521/zip/imm4521.zip.

To group $N$ objects $\mathbf{X} = \{\mathbf{x}_n\}_{n=1}^{N}$ we can represent the similarity or dissimilarity between the datapoints in a symmetrical $N \times N$ *proximity matrix* $\mathbf{D} = \{d_{i,i'}\}_{i,i'=1}^{N}$, where each element $d_{i,i'}$ holds the proximity between the objects $\mathbf{x}_i$ and $\mathbf{x}_{i'}$ and the diagonal $d_{ii} = 0$. If an object has several attributes (size, color, price etc.) is can be convenient to weight these attributes individually to balance their influence. Assuming $b$ attributes we can express the elements in $\mathbf{D}$ as

$$d_{i,i'} = \sum_{j=1}^{b} w_j \cdot d_j(x_{ij}, x_{i'j}) \tag{2.10}$$

where $\sum_{j=1}^{b} w_j = 1$ and $d_j$ denotes the attribute similarity measure. It is important to note that equal weights for all attributes does not necessarily mean equal influence, since it depends on the relative contribution to the average object dissimilarity measure. It can be shown that setting the weights to $w_j = 1/\hat{d}_j$ gives equal influence to all attributes, where $\hat{d}_j$ is the average dissimilarity of the j'th attribute given by $\hat{d}_j = 1/N^2 \sum_i \sum_{i'} d_j(x_{ij}, x_{i'j})$ [9]. A common similarity measure is the *squared Euclidian distance* given by

$$d_{i,i'} = \sum_{j=1}^{p} w_j \cdot (x_{ij} - x_{i'j})^2 \tag{2.11}$$

where the average dissimilarity of the attributes become $\hat{d}_j = 2 \cdot \mathrm{var}_j$. Thus the influence of the j'th attribute is proportional to its variance over the dataset.

One of the most simple clustering methods using the squared Euclidian distance as a similarity measure is the iterative descent type *K-Means* algorithm. The basic approach is to associate each $M$ dimensional datapoint $\mathbf{x}_n$ to one and only one cluster $k$ with mean vector $\mu_k = \{\mu_{m,k}\}_{m=1}^{M}$ and based on these associations the mean of each cluster $\mu_k$ is re-calculated in an iterative process. The similarity measure for the K-Means can be written as

$$d(\mathbf{x}_n, \mu_k) = \frac{1}{M} \sum_{m=1}^{M} \left(x_{m,n} - \mu_{m,k}\right)^2 = \frac{1}{M} \parallel \mathbf{x}_n - \mu_k \parallel^2 \tag{2.12}$$

This approach forms the iterative loop and is continued until the cluster associations do not change. The pseudocode for the K-Means algorithm can be illustrated very simple and is shown is table 2.1 (For `MATLAB` implementation refer to appendix B.2).

```
1. Initialize the $K$ cluster means to $\mu_k$, where $k = \{1, 2, \dots, K\}$
2. Calculate squared Euclidian distance between each datapoint $\mathbf{x}_n$ and cluster
   means $\mu_k$ as defined in (2.12)
3. Associate each datapoint $\mathbf{x}_n$ to the cluster with the shortest distance
       $\phi_k = \{\mathbf{x}_n \mid \operatorname*{argmin}_{\forall n} d(\mathbf{x}_n, \mu_k)\}$
4. Re-calculate the cluster means, based on the $N'$ associated datapoints $\mathbf{x}_n$
       $\mu_k = \frac{1}{N'} \sum_{n \in \phi_k} \mathbf{x}_n$
5. Repeat steps 2 - 4 until there are no change in cluster associations.
```

*Table 2.1:* Pseudocode for the K-Means algorithm

Here $\phi_k$ denotes the subset of datapoints assigned to cluster $k$, e.g. cluster $1$ may have 3 datapoints assigned as $\phi_1 = \{\mathbf{x}_2, \mathbf{x}_5, \mathbf{x}_9\}$. As each datapoint $\mathbf{x}_n$ is associated to one and only one cluster given by $\phi_k$, the K-Means algorithm is a *hard decision* type. In contrast the *Mixture of Gaussians* is a *soft decision* type, as we shall see later. The K-Means algorithm only estimates the cluster means $\mu_k$, so in case of modelling a Gaussian distribution the corresponding variance $\sigma_k^2$ can be estimated based on the cluster associations by $\sigma_k^2 = \frac{1}{N'} \sum_{n \in \phi_k} (x_n - \mu_k)^2$.

Model Parameters

The amount of clusters $K$ in K-Means is an essential model parameter. If $K$ is too low we might obtain an *underfit* and suffer from a bias failing to adapt to the underlying structure of the dataset $\mathbf{X}$. If $K$ is too high we risk achieving an *overfit*, where each cluster has only one association in the extreme case. Figure 2.1 gives an illustration of the two cases.

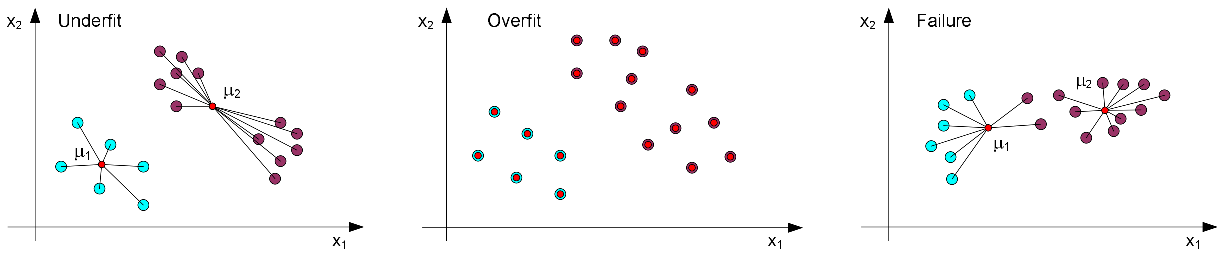

*Figure 2.1: left:* Case of underfitted clustering. *Middle:* Case of overfitted clustering. *Right*: Case of failed clustering.

The choice of $K$ is therefore essential and several techniques exists to find the optimal number such as *GAP statistics*, not discussed here. Refer to [9] for description of GAP statistics.

Cluster Initialization

The $K$ clusters can be initialized in different approaches. Initially we can generate $K$ random unity length vectors with uniform distribution with no relation to the actual dataset $\mathbf{X}$. Visually this would be a noisy image patch. This however suffers from the risk of achieving unassigned clusters, since they might be initialized far from the actual dataset and hence be overrun by competing cluster assignments. Any unassigned clusters can however quickly be identified from any of the initial noise image patches.

A different approach is to incorporate the dataset $\mathbf{X}$ in the initialization and choose $K$ random datapoints as clusters. This has the advantage of being in the vicinity of the dataset from the first iteration and thus decreases the number of iterations needed.

Finally the initial clusters can be generated from a Gaussian distribution using the mean of the dataset $D$ and choosing an initial width of the variance.

Limitations

The K-means algorithm being a competitive learning method has some relatively fatal limitations in certain cases. As it is a hard-decision type, since it only assigns datapoints to one cluster by $\phi_k$, it does not consider any 2nd order variance information and can therefore not take any shape or size of a cluster into account. An example where this becomes a limitation is shown in the right illustration in figure 2.1. Due to the shape of the two clusters two datapoints from the right cluster has incorrectly been assigned to the left cluster, because of the shorter distance yielding a bad grouping.

In general datapoints located near the border of two or more clusters should arguably play a partial role for the surrounding clusters, but since datapoints within a cluster has equal weights independent of their individual distance, these cases arise [19]. The mixture of Gaussians model using soft assignment instead is by far more robust to this type of failing cases and will be presented in section 2.4.

A generative model will later be presented based on the K-means algorithm in section 3.3. Next we will describe clustering from a probabilistic point of view in terms of density estimation.

## 2.3 Probabilistic Density Estimation

In the previous section we saw how the clustering K-Means algorithm has the ability to cluster data with few simple iterating steps, due to its hard-assignment of data to the individual clusters. In this section we introduce the class of probabilistic models for density estimation as an improved alternative using soft-assignment of datapoints instead. In terms of density estimation this means any observed data $\mathbf{X} = \{\mathbf{x}_n\}_{n=1}^{N}$ is modelled using a probabilistic model $\mathcal{M}$ with a set of parameters $\boldsymbol{\Theta}$, as the name implies. The Gaussian mixture model (MoG) and the mixture of factor analyzers (MFA) are examples of such probabilistic model to name a few.

First we will introduce a set of preliminary statistical concepts or tools for estimating model parameters $\Theta$ and then present a set of methods used for density estimation in relation to our cognitive generative model.

### 2.3.1 Maximum Likelihood Estimation

For a given model $\mathcal{M}$ the objective is to estimate the parameters $\boldsymbol{\Theta}$, such that the model best describe the given observations $\mathbf{X}$, that is to maximize the probability $p(\boldsymbol{\Theta}|\mathbf{X})$. If we apply *bayes'* rule, we can rewrite the posterior $p(\boldsymbol{\Theta}|\mathbf{X})$ into

$$p(\boldsymbol{\Theta}|\mathbf{X}) = \frac{p(\mathbf{X}|\boldsymbol{\Theta})p(\boldsymbol{\Theta})}{p(\mathbf{X})} \tag{2.13}$$

where $p(\mathbf{X}|\boldsymbol{\Theta})$ is the *likelihood* that the model with parameters $\boldsymbol{\Theta}$ are responsible for generating $\mathbf{X}$. The probability $p(\boldsymbol{\Theta})$ is the *prior* information on the model parameters and $p(\mathbf{X})$ denotes the *evidence*. Several approaches exist to determine an optimal set of parameters $\boldsymbol{\Theta}^*$ using different assumptions. In the *Maximum Likelihood* (ML) method the unknown parameters $\boldsymbol{\Theta}$ are assumed uniform distributed, i.e. no prior information are available on them. Since the model parameters $\boldsymbol{\Theta}$ are not dependent on $p(\mathbf{X})$, maximizing the likelihood is equivalent to maximizing the posterior $p(\boldsymbol{\Theta}|\mathbf{X})$. In the ML method the parameters $\boldsymbol{\Theta}$ are chosen such that they maximize the likelihood function $p(\mathbf{X}|\boldsymbol{\Theta})$ expressed by

$$p(\mathbf{X}|\boldsymbol{\Theta}) = p(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N|\boldsymbol{\Theta}) \tag{2.14}$$

As many density functions are based on exponential terms it is convenient to use the logarithm of the likelihood or the *log-Likelihood* $\mathcal{L}$ instead. This has no impact on the maximization of the likelihood, since the logarithm is a monotonic function. If we further assume the observed data samples $\mathbf{x}_n$ are all independent and identically distributed (IID) , we can express the log-likelihood $\mathcal{L}$ as the sum of the individual log-likelihoods by

$$\mathcal{L}(\boldsymbol{\Theta}) = \ln p(\mathbf{X}|\boldsymbol{\Theta}) = \sum_{n=1}^{N} \ln p(\mathbf{x}_n|\boldsymbol{\Theta}) \tag{2.15}$$

To maximize the likelihood the 1st order derivative is set to zero, $\frac{\partial}{\partial \boldsymbol{\Theta}} \ln p(\mathbf{X}|\boldsymbol{\Theta}) = 0$ and the parameters $\boldsymbol{\Theta}$ can be estimated as

$$\boldsymbol{\Theta}^* = \underset{\Theta}{\operatorname{argmax}} \ \ln p(\mathbf{X}|\boldsymbol{\Theta}) \tag{2.16}$$

With this approach the model parameters $\boldsymbol{\Theta}$ can be found analytically for simple likelihood functions. However as the likelihood function can be a complex function, which does not derivate easy, other iterative approaches can be used. In section 2.3.3 the EM-algorithm is introduced as an iterative approach for maximum likelihood estimation used widely to fit model parameters in density estimation.

A different challenge in modelling is the choice of the dimension or size of the model parameters, $\boldsymbol{\Theta}$. In the next subsection the Bayes Information Criteria is shortly presented as a statistical tool for model selection.

### 2.3.2 Model Selection and Bayes Information Criteria

Using the maximum likelihood approach to estimate model parameters for a given observed dataset $\mathbf{X}$ does not necessarily guarantee the optimal set of parameters, as the likelihood is also dependent on the dimension of the model parameters or the free parameters, denoted $k$. Selecting the right model is therefore essential in obtaining a representative model.

Choosing a low dimension of the model, we risk failing to capture the underlying structure of the data $\mathbf{X}$ and suffer from a bias. Conversely a high dimension of the model allows greater flexibility for fitting model parameters resulting in a higher likelihood, but also increases the risk of achieving an overfit to the training data. This is the basic *Bias/Variance trade-off* discussed shortly in appendix A.6.

To find an optimal number of independent model parameters $k$ the *Bayesian Information Criterion* (BIC) can be used to compare different models [27]. The BIC is derived using Bayes theorem [27] and can be expressed as

$$\text{BIC} = -2\mathcal{L}_{max} + k\ln(N) \tag{2.17}$$

where $\mathcal{L}_{max}$ is the log of the max. likelihood of the comparing models and $N$ is the number of observations. Using the criteria in (2.17) the model with the lowest BIC should be the preferred selection. This shows how high dimensional models (large k) are penalized and should only be accepted if provide highly likely descriptions of the data. By changing the sign of the expression, eq. (2.17) can be seen as seeking the highest likelihood pr. model dimension and observation [27].

Here it is important to note that choosing an optimal model is also context dependent. The optimal size for a cognitive generative model can not necessarily be evaluated through the BIC. For this type of model, the optimal dimension may only be evaluated manually, as we shall see later.

### 2.3.3 Expectation-Maximization Algorithm

The EM-algorithm being a iterative method is not based on gradient descent (1st order) or Newton's method (2nd order), which both are common approaches in solving non-linear optimization problems.

In the EM-algorithm the strategy is to make a local approximation, which is a lower-bound on the likelihood function and progressively improve the bound (i.e. increase the bound) through iterations until the EM has converged to a maximum. The algorithm then includes two steps, the "E"-step, which computes the bound and an "M"-step, which maximizes the bound. This will be more evident in the following sections.

<u>Latent Variable Models</u>

Any observed dataset $\mathbf{x}$ does not necessarily reveal the actual structure of the model as for instance mixture models. The data $\mathbf{x}$ is hence assumed *incomplete* (or has *missing* data) and it therefore becomes convenient to introduce hidden or *latent* variables $\mathbf{h}$ for the model to better describe the incomplete observed data $\mathbf{x}$. The latent variables can for instance be discrete component labels representing a sort of imaginary class labels for the observed data. The log-likelihood for the data as defined in (2.15) is thus redefined as the incomplete log-likelihood as

$$\mathcal{L}_{inc}(\mathbf{\Theta}) = \sum_n \ln p(\mathbf{x}_n|\mathbf{\Theta}) = \sum_n \ln \int p(\mathbf{x}_n|\mathbf{H},\mathbf{\Theta})\, \mathrm{d}\mathbf{H} \tag{2.18}$$

where $\mathbf{H} = \{\mathbf{h}_n\}_{n=1}^N$ is the latent random variable invoked. By introducing the latent variables $\mathbf{h}_n$, it can be shown that the incomplete log-likelihood $\mathcal{L}_{inc}(\mathbf{\Theta})$ is upper-bounded by the function $\mathcal{F}(p_{\mathbf{h}},\mathbf{\Theta})$ (see appendix A.2 for derivation and proof) defined as

$$\mathcal{F}(p_{\mathbf{h}},\mathbf{\Theta}) \triangleq \sum_n \int p_{\mathbf{h}}(\mathbf{h}_n) \ln \frac{p(\mathbf{x}_n,\mathbf{h}_n|\mathbf{\Theta})}{p_{\mathbf{h}}(\mathbf{h}_n)}\, \mathrm{d}\mathbf{h}_n \tag{2.19}$$

If we define the *complete* log-likelihood by $\mathcal{L}_c(\mathbf{\Theta}) = \ln p(\mathbf{X},\mathbf{H}|\mathbf{\Theta})$, we can rewrite (2.19) (see appendix A.2) into

$$\mathcal{F}(p_{\mathbf{h}}, \mathbf{\Theta}) = \mathcal{E}_{\mathbf{h}}\{\mathcal{L}_c(\mathbf{\Theta})\} - \sum_n \mathcal{H}(\mathbf{h}) \tag{2.20}$$

where the first term is the expected value of the complete-data log-likelihood with respect to the latent variables $\mathbf{h}$ and the last term inside the sum is the entropy of $\mathbf{h}$, denoted $\mathcal{H}(\mathbf{h})$ (refer to appendix A.3).

The iterative *Expectation/Maximization*-algorithm (EM) alternates between maximizing $\mathcal{F}(p_{\mathbf{h}}, \mathbf{\Theta})$ wrt. $p_{\mathbf{h}}$ and $\mathbf{\Theta}$ respectively. Here it is important to note that only the first term in (2.20) is dependent on the model parameters $\mathbf{\Theta}$. This will simplify the computations of the steps in the EM-algorithm, outlined below.

$$\mathbf{E-step}: \; p_{\mathbf{h}_n}^{(m+1)} \leftarrow \underset{p_{\mathbf{h}_n}}{\operatorname{argmax}} \; \mathcal{F}(p_{\mathbf{h}}^{(m)}, \mathbf{\Theta}^{(m)}), \quad \forall n \tag{2.21}$$

$$\mathbf{M-step}: \; \mathbf{\Theta}^{(m+1)} \leftarrow \underset{\mathbf{\Theta}}{\operatorname{argmax}} \; \mathcal{F}(p_{\mathbf{h}_n}^{(m+1)}, \mathbf{\Theta}^{(m)}) \tag{2.22}$$

$$\Rightarrow \; \mathbf{\Theta}^{(m+1)} \leftarrow \underset{\mathbf{\Theta}}{\operatorname{argmax}} \; \mathcal{E}_{\mathbf{h}}\{\mathcal{L}_c(\mathbf{\Theta}^{(m)})\} \tag{2.23}$$

In the "M"-step the expected value of the parameters $\mathbf{\Theta}$ is maximized with constant latent variables $\mathbf{H}$. In the "E"-step the expected value of the log-likelihood $\mathcal{L}_c(\mathbf{\Theta})$ wrt. the latent variables $\mathbf{H}$ is estimated while keeping the parameters $\mathbf{\Theta}$ constant. The expectation in the "E"-step is exactly maximized when the bound becomes an equality, i.e. $\mathcal{L}_{inc}(\mathbf{\Theta}) = \mathcal{F}(p_{\mathbf{h}}, \mathbf{\Theta})$. This occurs when the two distributions $p_{\mathbf{h}_n}(\mathbf{h}_n) = p(\mathbf{h}_n|\mathbf{x}_n, \mathbf{\Theta})$ and becomes evident from (2.20).

$$\mathcal{F}(p_{\mathbf{h}}, \mathbf{\Theta}) = \sum_n \int p_{\mathbf{h}}(\mathbf{h}_n) \ln \frac{p(\mathbf{x}_n, \mathbf{h}_n|\mathbf{\Theta})}{p(\mathbf{h}_n|\mathbf{x}_n, \mathbf{\Theta})} \, \mathrm{d}\mathbf{h}_n \tag{2.24}$$

$$= \sum_n \int p_{\mathbf{h}}(\mathbf{h}_n) \ln p(\mathbf{x}_n|\mathbf{\Theta}) \, \mathrm{d}\mathbf{h}_n \tag{2.25}$$

$$= \sum_n \ln p(\mathbf{x}_n|\mathbf{\Theta}) \int p_{\mathbf{h}}(\mathbf{h}_n) \, \mathrm{d}\mathbf{h}_n \tag{2.26}$$

$$= \sum_n \ln p(\mathbf{x}_n|\mathbf{\Theta}) \tag{2.27}$$

$$= \mathcal{L}_{inc}(\mathbf{\Theta}) \tag{2.28}$$

This can also be seen from (A.14) in appendix A.2, where the KL divergence becomes zero for $p_{\mathbf{h}}(\mathbf{h}_n) = p(\mathbf{h}_n|\mathbf{x}_n, \mathbf{\Theta})$, i.e. $\mathrm{KL}(p_{\mathbf{h}}(\mathbf{h}_n) \mid\mid p_{\mathbf{h}}(\mathbf{h}_n)) = 0$.

The "E"- and the "M"-steps can together be interpreted as shown in figure 2.2. Here the "E"-step forms the bound $\mathcal{F}(p_{\mathbf{h}}, \mathbf{\Theta})$, which is tangent to the log-likelihood $\mathcal{L}_{inc}(\mathbf{\Theta})$ and the "M"-step then subsequently maximizes this bound as shown on the figure. In addition the steps can also be seen as coordinate ascent in likelihood-space as illustrated in figure 2.3, where each step can only be made either horizontally or vertically until a maximum is found. The maximization of the bound in the "E"-step cannot be seen in this figure, since the surface curves illustrate $\mathcal{L}_c(\mathbf{\Theta})$ and not the bound $\mathcal{F}(p_{\mathbf{h}}, \mathbf{\Theta})$.

Prior to any "M"-step $\mathcal{F}(p_{\mathbf{h}}, \mathbf{\Theta}) = \mathcal{L}_{inc}(\mathbf{\Theta})$ and since the parameters $\mathbf{\Theta}$ are assumed constant, we are guaranteed not to decrease the likelihood $\mathcal{L}_{inc}(\mathbf{\Theta})$. This correspond to minimizing the Kullback-Leibler divergence $\mathrm{KL}\big(p_{\mathbf{h}}(\mathbf{h}_n) \mid\mid p(\mathbf{h}_n|\mathbf{x}_n, \mathbf{\Theta})\big)$ in (A.14). This has also been shown by Neal & Hinton [23] and Hastie, Tibshirani & Friedman [9].

Eventhough the EM-algorithm is ensured to maximize the likelihood $\mathcal{L}_c(\mathbf{\Theta})$, it is susceptible to local maxima and thus not guaranteed to find the global maximum. In most cases it is actually not desired to converge to a global maximum, since such a solution is considered an overfit. For the Gaussian mixture model this manifests into mixture component fitted to single datapoints with variances close to zero. This will become clearer in the later sections.

Having described a general framework for the EM-algorithm, we now present the Gaussian mixture model and apply the EM-algorithm to derive a set of iterating steps for this type of model.

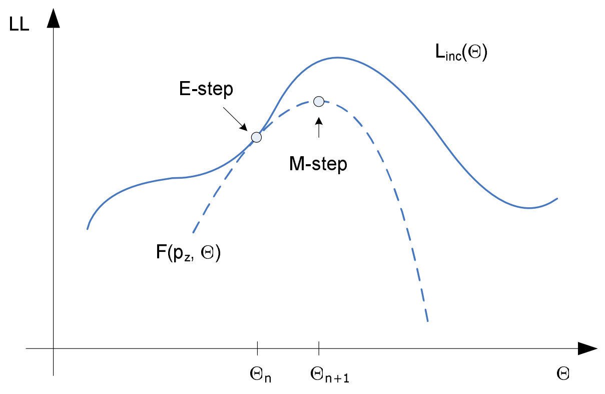

*Figure 2.2:* Conceptual illustration of the EM-steps. The "E"-step estimates the upper-bound on the likelihood function $\mathcal{F}(p_{\mathbf{h}}, \mathbf{\Theta})$ and the "M"-step maximizes that bound.

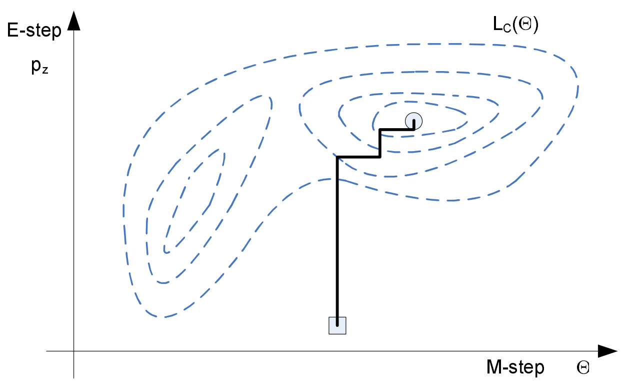

*Figure 2.3:* This figure illustrates the coordinate ascent of the EM-algorithm, iterating between the "E"- and the "M"-step.

## 2.4 Gaussian Mixture Model

The *Gaussian mixture model* or the *Mixture of Gaussians* (MoG) is a latent variable model widely used in density modelling due to its simplicity. If we assume $N$ observed data samples $\mathbf{X} = \{\mathbf{x}_n\}_{n=1}^{N}$, where $\mathbf{x}$ is a random $d$ dimensional variable, the MoG model expresses the probability density $p(\mathbf{x})$ as a linear combination of $K$ Gaussian mixtures on the general form

$$p(\mathbf{x}) = \sum_{k=1}^{K} p(\mathbf{x}|k)p(k) = \sum_{k=1}^{K} \alpha_k \cdot p(\mathbf{x}|k) \tag{2.29}$$

where $\alpha_k = p(k)$ is the component prior summing to 1 for all $k$, i.e. $\sum_k \alpha_k = 1$ and $p(\mathbf{x}|k)$ denotes the Gaussian distribution function given by

$$p(\mathbf{x}|k) = \frac{1}{\sqrt{(2\pi)^d |\mathbf{\Sigma}_k|}} \exp\Big(-\frac{1}{2}(\mathbf{x}-\mu_k)^T \mathbf{\Sigma}_k^{-1}(\mathbf{x}-\mu_k)\Big) \tag{2.30}$$

where the 1st order mean $\mu_k$ and the 2nd order covariance matrix $\mathbf{\Sigma}_k$ are the parameters for the $k$'th Gaussian component. Thus all the MoG model parameters become $\mathbf{\Theta} = \{\alpha_k, \mu_k, \mathbf{\Sigma}_k\}_{k=1}^{K}$.

In this model we assume the observed each data sample $\mathbf{x}_n$ is generated from one of the Gaussian component $p(\mathbf{x}|k)$, however the information of which mixture component was actually responsible for the generation is not directly available. This information must be inferred from the observables and for this purpose we introduce the $K$ dimensional binary latent variable $\mathbf{h}_n = \{h_{k,n}\}_{k=1}^{K}$ which expresses which component $k$ is responsible for generating the n'th sample $\mathbf{x}_n$. That is we define $h_{k,n}$ to hold a boolean 1 at the $k$'th position for the $n$'th sample if the $k$'th component actually generated $\mathbf{x}_n$ [3] [4], e.g.

$$\mathbf{h}_n = \{00\ldots010\ldots00\}^T \tag{2.31}$$

where $p(h_{k,n} = 1) = \alpha_k$ in (2.29) and $\mathbf{H} = \{\mathbf{h}_n\}_{n=1}^{N}$, becomes an $K \times N$ matrix. This notation allows the number of samples associated by component $k$ to be found as $N_k = \sum_n h_{k,n}$.

As the EM-algorithm assumes the presence of latent or missing variables $\mathbf{H}$ it becomes a obvious choice in estimating the parameters $\mathbf{\Theta}$ of the MoG model. The complete log-likelihood based on $\mathbf{h}$ can now be written as

$$\mathcal{L}_c(\mathbf{\Theta}) = \ln p(\mathbf{X}, \mathbf{H}|\mathbf{\Theta}) \tag{2.32}$$

$$= \ln \prod_n^N p(\mathbf{x}_n, \mathbf{h}_n|\mathbf{\Theta}) \tag{2.33}$$

$$= \ln \prod_n^N \prod_k^K \Big(p(\mathbf{x}_n|h_{k,n}=1, \mathbf{\Theta})p(h_{k,n}=1)\Big)^{h_{k,n}} \tag{2.34}$$

$$= \sum_n^N \sum_k^K h_{k,n}\Big(\ln p(\mathbf{x}_n|h_{k,n}=1,\mathbf{\Theta}) + \ln \alpha_k\Big) \tag{2.35}$$

where $h_{k,n}$ in the exponent in (2.34) serves to only select or activate the mixture component $k$ responsible for generating $\mathbf{x}_n$. The expression in (2.35) form the basis in the derivation of the update equations for the EM-algorithm, i.e. the "E"- and "M"-steps.

M-step

As mentioned the "M"-step maximizes the expectation of the complete log-likelihood function $\mathcal{E}_{\mathbf{h}}\{\mathcal{L}_c(\mathbf{\Theta})\}$ under the assumption that the latent variables $\mathbf{h}_n$ are known. This expectation $\mathcal{E}_{\mathbf{h}}\{\mathcal{L}_c(\mathbf{\Theta})\}$ can be expressed based on (2.35) as

$$\mathcal{E}_{\mathbf{h}}\{\mathcal{L}_c(\mathbf{\Theta})\} = \sum_n^N \sum_k^K w_{k,n}\big(\ln p(\mathbf{x}_n|h_{k,n}=1,\mathbf{\Theta}) + \ln \alpha_k\big) \tag{2.36}$$

where $w_{k,n} = \mathcal{E}_{\mathbf{h}}\{h_{k,n}\}$ denotes the responsibility or weight of the $k$'th mixture component in generating sample $n$. From (2.36) we can derive a set of update equation for the model parameters $\mathbf{\Theta} = \{\alpha_k, \mu_k, \mathbf{\Sigma}_k\}_{k=1}^K$ by differentiating $\mathcal{E}_{\mathbf{h}}\{\mathcal{L}_c(\mathbf{\Theta})\}$ wrt. to the different parameters. Initially we derive the update for the mean $\mu_k$.

$$\frac{\partial \mathcal{E}_{\mathbf{h}}\{\mathcal{L}_c(\mathbf{\Theta})\}}{\partial \mu_k} = \sum_n^N w_{k,n} \frac{\partial}{\partial \mu_k} \ln p(\mathbf{x}_n|h_{k,n}=1,\mathbf{\Theta}) = 0 \tag{2.37}$$

where $p(\mathbf{x}_n|h_{k,n}=1,\mathbf{\Theta})$ is the Gaussian distribution given by (2.30). As this expression only includes dependency to $\mu_k$ in the quadratic form inside the exponential term, the partial derivative becomes

$$\frac{\partial}{\partial \mu_k} \ln p(\mathbf{x}_n|h_{k,n}=1,\mathbf{\Theta}) = (\mathbf{x}_n - \mu_k)^T \mathbf{\Sigma}_k \tag{2.38}$$

This is substituted into (2.37) and we can isolate the mean $\mu_k$

$$\sum_n^N w_{k,n}(\mathbf{x}_n - \mu_k)^T \mathbf{\Sigma}_k = 0 \qquad \Leftrightarrow \tag{2.39}$$

$$\mu_k = \frac{\sum_n^N w_{k,n}\mathbf{x}_n}{\sum_n^N w_{k,n}} \tag{2.40}$$

This gives the intuitive results that the mean $\mu_k$ is estimated as an average of weighted datapoints $\mathbf{x}_n$. Similarly the covariance matrix $\mathbf{\Sigma}_k$ can be found by differentiating (2.36) wrt. $\mathbf{\Sigma}_k$ and by the same manipulations we get

$$\mathbf{\Sigma}_k = \frac{\sum_n^N w_{k,n}(\mathbf{x}_n - \mu_k)(\mathbf{x}_n - \mu_k)^T}{\sum_n^N w_{k,n}} \tag{2.41}$$

This also shows that the covariance $\mathbf{\Sigma}_k$ is estimated over weighted zero-mean datapoints. To estimate the prior $\alpha_k$ we need to include a few constraints. As $\alpha_k$ is a probability it must be between $0$ and $1$ and must sum to $1$

for all $k$, i.e. $\sum_k \alpha_k = 1$. These constraints are obeyed by augmenting (2.36) with a *Lagrange* multiplier (refer to [20] and [4] for a description of constrained optimization), such that

$$\mathcal{C}(\mathbf{\Theta}) = \mathcal{E}_{\mathbf{h}}\{\mathcal{L}_c(\mathbf{\Theta})\} - \lambda\big(\sum_k^K \alpha_k - 1\big) \tag{2.42}$$

Setting the 1st derivative wrt. $\alpha_k$ to zero, we get

$$\begin{aligned}
\frac{\partial}{\partial \alpha_k}\mathcal{C}(\mathbf{\Theta}) &= \frac{\partial}{\partial \alpha_k}\Big[\mathcal{E}_{\mathbf{h}}\{\mathcal{L}_c(\mathbf{\Theta})\} - \lambda\big(\sum_k^K \alpha_k - 1\big)\Big] && (2.43)\\
&= \frac{\partial}{\partial \alpha_k}\left[\sum_n^N \sum_k^K w_{k,n}\big(\ln p(\mathbf{x}_n|h_{k,n}=1,\mathbf{\Theta}) + \ln \alpha_k\big) - \lambda\big(\sum_k^K \alpha_k - 1\big)\right] && (2.44)\\
&= \sum_{n=1}^N \frac{1}{\alpha_k} w_{k,n} - \lambda = 0 && (2.45)
\end{aligned}$$

Summing over all values of $k$ and using $\sum_k \alpha_k = 1$ and $\sum_k w_{k,n} = 1$, we obtain $\lambda = N$. From this we can isolate $\alpha_k$

$$\alpha_k = \frac{1}{N}\sum_{n=1}^N w_{k,n} \tag{2.46}$$

This intuitive result shows that the prior information for each component is based on how large the responsibility is for each component.

E-step

In the "E"-step the latent variables $h$ are estimated under the assumption that the model parameters $\mathbf{\Theta}$ are constant or known. Previously the "E"-step was shown in (2.24) - (2.28) to be maximized when $p_{\mathbf{h}}(\mathbf{h}_n) = p(\mathbf{h}_n|\mathbf{x}_n,\mathbf{\Theta})$. Hence the expectation of the latent variables $w_{k,n}$ should be taken wrt. the conditional probability $p(\mathbf{h}_n|\mathbf{x}_n,\mathbf{\Theta})$. We therefore compute the weights $w_{k,n}$ by averaging over all possible outcomes.

$$\begin{aligned}
w_{k,n} = \mathcal{E}\{h_{k,n}\} &= \int h_{k,n}\, p(h_{k,n}|\mathbf{x}_n,\mathbf{\Theta})\, \mathrm{d}h_{k,n} && (2.47)\\
&= 1\cdot p(h_{k,n}=1|\mathbf{x}_n,\mathbf{\Theta}) + 0\cdot p(h_{k,n}=0|\mathbf{x}_n,\mathbf{\Theta}) && (2.48)\\
&= p(h_{k,n}=1|\mathbf{x}_n,\mathbf{\Theta}) && (2.49)
\end{aligned}$$

From bayes' rule we can rewrite (2.49) into

$$w_{k,n} = \frac{p(\mathbf{x}_n|h_{k,n}=1,\mathbf{\Theta})\cdot\alpha_k}{\sum_{k'}^K p(\mathbf{x}_n|h_{k',n}=1,\mathbf{\Theta})\cdot\alpha_{k'}} \tag{2.50}$$

This final expression gives the intuitive appealing result that the responsibility for each sample $\mathbf{x}_n$ is based on the gradual influence from each weighted Gaussian mixture component.

As discussed earlier the EM-algorithm is susceptible to any maximum, local or global. In the special case of converging to a global maximum, the MoG model overfits to the data, where each Gaussian component represents a single datapoint with variance close to zero. This is not a desired case as the MoG model has failed to capture the underlying structure of the dataset. Controlling the convergence of the likelihood is therefore evident and for the MoG model a few restriction can be made on the complexity of the model to prevent potential overfit. These will be presented in the following section.

### 2.4.1 Model Complexity

The Gaussian kernel function $p(\mathbf{x}|k)$ defined in (2.30) can as mentioned be described completely by the 1st order mean $\mu_k$ and 2nd order variance $\boldsymbol{\Sigma}_k$. Despite only having two parameter the complexity of the kernel function can be controlled by restricting the flexibility of the variance or limiting the number of free parameters. In our analysis we will focus on 3 different variances denoted *isotropical*, *diagonal* and *full* illustrated in figure 2.4.

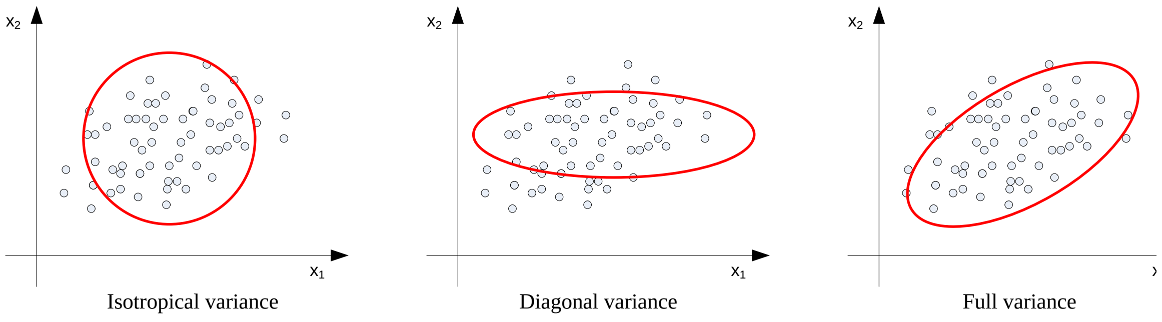

*Figure 2.4:* The 3 different restrictions of variance.

Isotropical variance

The isotropical Gaussian function has a variance defined as $\boldsymbol{\Sigma}_k^{(iso)} = \sigma_k^2\mathcal{I}$, where $\sigma_k^2$ is a scalar, i.e. a constant diagonal variance $\sigma_k^2$ and can be estimated for each component by

$$\sigma_k^2 = \frac{1}{d}\sum_{i=1}^{d}\frac{\sum_{n=1}^{N} w_{k,n}(\mathbf{x}_{i,n}-\mu_{i,k})^2}{\sum_{n=1}^{N} w_{k,n}} \tag{2.51}$$

Thus any covariance information is not included as the off-diagonal element of $\boldsymbol{\Sigma}_k^{(iso)}$ are forced to zero. Visually this forms a circular shape shown in the left figure in 2.4. This also limits the number of parameters to estimate to $K$.

Given a dataset $\mathbf{X}$ it can be seen from the definition how this isotropical variance can limit the model from overfitting to $\mathbf{X}$, i.e. avoid adapting to specific datapoints. In contrast this type of variance can also induce a bias to the model. This is the basic bias/variance trade-off.

Diagonal variance

From the isotropical model, the variance can be expanded to diagonal variance allowing greater flexibility. A diagonal covariance matrix is defined only by its diagonal elements, i.e. $\boldsymbol{\Sigma}_k^{(diag)} = \sigma_k^2\mathcal{I}$, where $\sigma_k^2$ is a $d$ dimensional vector and is estimated by

$$\sigma_k^2 = \frac{\sum_{n=1}^{N} w_{k,n}(\mathbf{x}_n-\mu_k)^2}{\sum_{n=1}^{N} w_{k,n}} \tag{2.52}$$

This type also assumes no correlation between the individual dimensions of $\mathbf{x}$ and thus does not take any covariance into account. This gives a elliptic shape shown in the middle illustration in figure 2.4. In addition the number of parameters to estimate is limited to $Kd$.

Full variance

Finally the full variance $\boldsymbol{\Sigma}_k^{(full)}$ has full flexibility and is defined for the 2 dimensional case in (A.3) in appendix A.1. It is clear that the full variance includes covariance information and thereby has full flexibility. In contrast to the isotropical variance, the full variance risk adapting too precisely to individual datapoints and thereby suffer from overfit. This is again the other side of the bias/variance trade-off.

The full information of the covariance included in $\mathbf{\Sigma}_k^{(full)}$ can be seen as a slight skew of the ellipse on the right illustration in figure 2.4. For the full covariance, the number of parameters to estimate is diagonal plus half the off-diagonal elements, expressed as $K(\frac{1}{2}(d^2 - d) + d) = Kd(\frac{1}{2}(d-1) + 1)$.

Summary

The "EM"-algorithm presented for the MoG model is implemented in MATLAB with all 3 types of covariance matrices and can be found in appendix B.2. In sections 3.4 & 3.5 we present a small set of illustrative simulations of the MoG model to view the performance in action.

In our analysis of the MoG model, we have assumed that the amount of samples $N$ is much larger than the dimension of the data $d$, i.e. $N >> d$. In cases where this is not true, the $N$ datapoints span only a low-dimensional subspace of $\mathcal{R}^d$. This leads to a poor estimation of the Gaussian mixture parameter $\mu_k$ and $\mathbf{\Sigma}_k$, in particular the covariance $\mathbf{\Sigma}_k$, which risk being singular. This is the curse of dimensionality.

In order to avoid poorly estimated covariance matrices and a potential overfit of the MoG model, the covariance matrix $\mathbf{\Sigma}_x$ can be restricted in 2 different ways, *isotropical* or *diagonal* variance, as discussed. However these restriction assume no covariance in the datapoints, which may limit the MoG modelling capabilities.

We therefore introduce a different mixture model, which includes a dimension reduction of the data in the modelling.

## 2.5 Mixture of Factor Analyzers

In unsupervised learning two of the major disciplines are clustering and dimension reduction. In appendix A.1 we discuss and present *Principal Component Analysis* as a tool for reducing dimensions of data. In this section we will present the *Mixture of Factor Analyzers* (MFA) combining both clustering and dimension reduction as an alternative to MoG with improved modelling properties of covariance information.

Instead of using the Gaussian distribution (2.30) as the kernel function in our mixture model, we employ a *factor analyzer* (FA), which both performs dimension reduction and Gaussian modelling with potentially fewer parameters. Initially these two steps could be conducted separately using PCA and MoG, but by combining them some of the drawbacks of PCA in terms of probabilistic modelling can be avoided. Unlike FA PCA does not define a proper density model for the data, as the cost of transforming a point is equal anywhere along the principal component subspace [7].

In addition different features may be correlated within different clusters, which means the metric for the dimension reduction may need to be different for different clusters. By including dimension reduction the process of cluster formation may become easier, since different clusters may appear more separated depending on the local metric [7].

Thus in combining clustering and dimension reduction we may achieve a synergetic effect and an improved model compared to MoG. Before we present the MFA, we proceed with describing *Factor Analysis* in general form and derive a set of update equations based on the "EM"-algorithm.

### 2.5.1 Factor Analysis

In factor analysis the $d$ dimensional random variable $\mathbf{x}$ is assumed Gaussian distributed and is linearly decomposed into lower $d'$ dimensional factors $\mathbf{z}$, where $d$ usually is much smaller than $d'$. The generative factor analyzer (FA) model is given by

$$\mathbf{x} = \mathbf{\Lambda}\mathbf{z} + \epsilon \tag{2.53}$$

where $\mathbf{\Lambda}$ is the $d \times d'$ *factor loading* matrix. The corresponding $d'$ dimensional factors $\mathbf{z}$ are assumed to have zero mean with unity variance, i.e. $\mathcal{N}(0, \mathcal{I})$. Finally $\epsilon$ is a $d$ dimensional random variable holding the residual uncorrelated noise and is hence assumed to have zero mean and a diagonal variance denoted by $\Psi$, i.e. $\mathcal{N}(0, \Psi)$. The diagonal variance for both $\mathbf{z}$ and $\epsilon$ is a vital assumption in factor analysis and leads to all correlation in the data $\mathbf{x}$ are modelled by factor loading matrix $\mathbf{\Lambda}$ and the factors $\mathbf{z}$ and that the residual independent noise are accounted for by $\epsilon$. This also make FA more robust to independent noise in dimension reduction compared to

PCA. In FA the $k$ factor-loading vectors in $\mathbf{\Lambda}$ plays the same role as the principal components for PCA, as they are the informative projections of the data [7].

Regular PCA as presented in section A.1 can also be derived as Probabilistic PCA (PPCA) on the same form as eq. (2.53) [29]. In PPCA the distribution of the independent noise $\epsilon$ is instead defined with isotropical variance, i.e. $\mathcal{N}(0, \sigma^2\mathcal{I})$, which accounts for the averaged variance 'lost' in the projection of data into subspace. Refer to [29] for a more detailed description of PPCA.

From the decomposition of $\mathbf{x}$ in (2.53) the covariance of $\mathbf{x}$ can easily be expressed as

$$\mathbf{\Sigma}_x = \mathcal{E}\{\mathbf{x}\mathbf{x}^T\} = \mathcal{E}\{(\mathbf{\Lambda}\mathbf{z}+\epsilon)(\mathbf{\Lambda}\mathbf{z}+\epsilon)^T\} = \mathcal{E}\{\mathbf{\Lambda}\mathbf{z}\mathbf{z}^T\mathbf{\Lambda}^T\} + \mathcal{E}\{\epsilon\epsilon^T\} = \mathbf{\Lambda}\mathcal{I}\mathbf{\Lambda}^T + \Psi = \mathbf{\Lambda}\mathbf{\Lambda}^T + \Psi \tag{2.54}$$

Thus the distribution of the data can be written as $p(\mathbf{x}|\mathbf{\Theta}) = \mathcal{N}(0, \mathbf{\Lambda}\mathbf{\Lambda}^T + \Psi)$, where $\mathbf{\Theta} = \{\mathbf{\Lambda}, \mathbf{\Psi}\}$ are the model parameters [4]. We can now express the log-likelihood of the parameters $\mathcal{L}_x(\mathbf{\Theta})$ based on (2.15) as $\mathcal{L}_x(\mathbf{\Theta}) = \sum_{n=1}^{N} \ln\, \mathcal{N}(\mathbf{0}, \mathbf{\Lambda}\mathbf{\Lambda}^T + \Psi)$. Estimating the model parameters from this log-likelihood is a hard problem to do in closed form, if even possible.

Instead we base the derivation on the "EM"-algorithm as presented in section 2.3.3. If we denote the factors $\mathbf{Z} = \{\mathbf{z}_n\}_{n=1}^N$ the latent variables, we can form the "M"-step from the complete log-likelihood of the model parameters from equation 2.23 given the observed dataset $\mathbf{X} = \{\mathbf{x}_n\}_{n=1}^N$, rewritten here for convenience

$$\mathcal{E}_{\mathbf{z}}\{\mathcal{L}_c(\mathbf{\Theta})\} = \mathcal{E}_{\mathbf{z}}\{\ln p(\mathbf{X}, \mathbf{Z}|\mathbf{\Theta})\} = \mathcal{E}_{\mathbf{z}}\{\ln p(\mathbf{X}|\mathbf{Z}, \mathbf{\Theta}) + \ln p(\mathbf{Z})\} \tag{2.55}$$

where $p(\mathbf{X}|\mathbf{Z}, \mathbf{\Theta})$ is the conditional Gaussian distribution of the data based on (2.53) with mean $\mu_{x|z} = \mathbf{\Lambda}\mathbf{z}$ and variance $\mathbf{\Sigma}_{x|z} = \mathbf{\Psi}$, i.e. $p(\mathbf{X}|\mathbf{Z}, \mathbf{\Theta}) = \mathcal{N}(\mathbf{\Lambda}\mathbf{z}, \mathbf{\Psi})$. From (2.55) we can neglect the last term $\ln p(\mathbf{Z})$, since it does not depend on the model parameters and thus we need to maximize

$$\mathcal{E}_{\mathbf{z}}\{\ln p(\mathbf{X}|\mathbf{Z}, \mathbf{\Theta})\} = \sum_{n=1}^{N} \mathcal{E}_{\mathbf{z}}\{\ln p(\mathbf{x}_n|\mathbf{z}_n, \mathbf{\Theta})\} \tag{2.56}$$

$$= \sum_{n=1}^{N} \mathcal{E}_{\mathbf{z}}\Big\{ \ln \frac{1}{\sqrt{(2\pi)^d\, |\mathbf{\Psi}|}} \exp\big( -\frac{1}{2}(\mathbf{x}_n - \mathbf{\Lambda}\mathbf{z}_n)^T\mathbf{\Psi}^{-1}(\mathbf{x}_n - \mathbf{\Lambda}\mathbf{z}_n)\big)\Big\} \tag{2.57}$$

$$= -\frac{Nd}{2}\ln(2\pi) - \frac{N}{2}\ln|\mathbf{\Psi}| - \sum_{n=1}^{N} \mathcal{E}_{\mathbf{z}}\Big\{\frac{1}{2}(\mathbf{x}_n - \mathbf{\Lambda}\mathbf{z}_n)^T\mathbf{\Psi}^{-1}(\mathbf{x}_n - \mathbf{\Lambda}\mathbf{z}_n)\Big\} \tag{2.58}$$

$$= c - \frac{N}{2}\ln|\mathbf{\Psi}| - \sum_{n=1}^{N} \mathcal{E}_{\mathbf{z}}\Big\{\frac{1}{2}\mathbf{x}_n^T\mathbf{\Psi}^{-1}\mathbf{x}_n - \mathbf{x}_n^T\mathbf{\Psi}^{-1}\mathbf{\Lambda}\mathbf{z} + \frac{1}{2}\mathbf{z}^T\mathbf{\Lambda}^T\mathbf{\Psi}^{-1}\mathbf{\Lambda}\mathbf{z}\Big\} \tag{2.59}$$

where $c$ is a constant independent of the model parameters. As the last term inside the expectation is a scalar, $\frac{1}{2}\mathbf{z}^T\mathbf{\Lambda}^T\mathbf{\Psi}^{-1}\mathbf{\Lambda}\mathbf{z} = C$, where $C \in \mathcal{R}$, we can use the facts that $\mathrm{tr}(C) = C$ and $\mathrm{tr}(AB) = \mathrm{tr}(BA)$ and rewrite (2.59) into

$$\mathcal{E}_{\mathbf{z}}\{\ln p(\mathbf{X}|\mathbf{Z}, \mathbf{\Theta})\} = c - \frac{N}{2}\ln|\mathbf{\Psi}| - \sum_{n=1}^{N}\Big(\frac{1}{2}\mathbf{x}_n^T\mathbf{\Psi}^{-1}\mathbf{x}_n - \mathbf{x}_n^T\mathbf{\Psi}^{-1}\mathbf{\Lambda}\mathcal{E}_{\mathbf{z}}\{\mathbf{z}|\mathbf{x}_n\} + \frac{1}{2}\mathrm{tr}\big(\mathbf{\Lambda}^T\mathbf{\Psi}^{-1}\mathbf{\Lambda}\mathcal{E}_{\mathbf{z}}\{\mathbf{z}\mathbf{z}^T|\mathbf{x}_n\}\big)\Big) \tag{2.60}$$

$$\triangleq Q(\mathbf{\Lambda}, \mathbf{\Psi}) \tag{2.61}$$

This expression is used for deriving the "M"-step for the update algorithm, shown below.

"M"-step :

If we denote the entire last expression $Q(\mathbf{\Lambda}, \mathbf{\Psi})$ in (2.60), we can derive the model parameters by setting the respective derivatives to zero. Initially we derive the update equation for the factor loading matrix $\mathbf{\Lambda}$.

[4]For PPCA, $p(\mathbf{x}|\mathbf{\Theta}) = \mathcal{N}(0, \mathbf{\Lambda}\mathbf{\Lambda}^T + \sigma^2\mathcal{I})$, where $\mathbf{\Theta} = \{\mathbf{\Lambda}, \sigma^{\mathbf{2}}\}$

$$\frac{\partial Q(\mathbf{\Lambda},\mathbf{\Psi})}{\partial \mathbf{\Lambda}^{(m)}} = -\sum_{n=1}^{N} \mathbf{\Psi}^{-1}\mathbf{x}_n \mathcal{E}_{\mathbf{z}}\{\mathbf{z}|\mathbf{x}_n\}^T + \sum_{n=1}^{N} \mathbf{\Psi}^{-1}\mathbf{\Lambda}^{(m+1)}\mathcal{E}_{\mathbf{z}}\{\mathbf{z}\mathbf{z}^T|\mathbf{x}_n\} = 0 \tag{2.62}$$

By setting this to zero, we can isolate for $\mathbf{\Lambda}^{(m+1)}$ and derive the update equation

$$\mathbf{\Lambda}^{(m+1)} = \left(\sum_{n=1}^{N} \mathbf{x}_n \mathcal{E}_{\mathbf{z}}\{\mathbf{z}|\mathbf{x}_n\}^T\right)\left(\sum_{n=1}^{N} \mathcal{E}_{\mathbf{z}}\{\mathbf{z}\mathbf{z}^T|\mathbf{x}_n\}\right)^{-1} \tag{2.63}$$

This show the intuitive result that the factor loading matrix $\mathbf{\Lambda}$ is based on 2nd order information of the factors $\mathbf{z}$. To find an update equation for $\mathbf{\Psi}$, we derive it through its inverse, $\mathbf{\Psi}^{-1}$. From (2.60) we get

$$\frac{\partial Q(\mathbf{\Lambda},\mathbf{\Psi})}{\partial \mathbf{\Psi}^{-1(m)}} = \frac{N}{2}\mathbf{\Psi}^{(m+1)} - \sum_{n=1}^{N}\left(\frac{1}{2}\mathbf{x}_n\mathbf{x}_n^T - \mathbf{\Lambda}^{(m+1)}\mathcal{E}_{\mathbf{z}}\{\mathbf{z}|\mathbf{x}_n\}\mathbf{x}_n^T + \frac{1}{2}\mathbf{\Lambda}^{(m+1)}\mathcal{E}_{\mathbf{z}}\{\mathbf{z}\mathbf{z}^T|\mathbf{x}_n\}\mathbf{\Lambda}^{T(m+1)}\right) = 0 \tag{2.64}$$

By substituting $\mathbf{\Lambda}^{(m+1)}$ with eq. (2.63) we get

$$\frac{N}{2}\mathbf{\Psi}^{(m+1)} = \sum_{n=1}^{N}\left(\frac{1}{2}\mathbf{x}_n\mathbf{x}_n^T - \mathbf{\Lambda}^{(m+1)}\mathcal{E}_{\mathbf{z}}\{\mathbf{z}|\mathbf{x}_n\}\mathbf{x}_n^T + \frac{1}{2}\mathbf{\Lambda}^{(m+1)}\mathbf{x}_n\mathcal{E}_{\mathbf{z}}\{\mathbf{z}^T|\mathbf{x}_n\}^T\right) \tag{2.65}$$

$$= \sum_{n=1}^{N}\left(\frac{1}{2}\mathbf{x}_n\mathbf{x}_n^T - \frac{1}{2}\mathbf{\Lambda}^{(m+1)}\mathcal{E}_{\mathbf{z}}\{\mathbf{z}|\mathbf{x}_n\}\mathbf{x}_n^T\right) \tag{2.66}$$

As $\mathbf{\Psi}$ must not contain any covariance information, we enforce a diagonal constraint and get

$$\mathbf{\Psi}^{(m+1)} = \frac{1}{N}\,\mathrm{diag}\left\{\sum_{n=1}^{N}\mathbf{x}_n\mathbf{x}_n^T - \mathbf{\Lambda}^{(m+1)}\mathcal{E}_{\mathbf{z}}\{\mathbf{z}|\mathbf{x}_n\}\mathbf{x}_n^T\right\} \tag{2.67}$$

In both update equations (2.63) and (2.67) there is a dependency to the latent variables $\mathcal{E}\{\mathbf{z}|\mathbf{x}_n\}$ and $\mathcal{E}\{\mathbf{z}\mathbf{z}^T|\mathbf{x}_n\}$ as they are assumed known or constant. In the "E"-step these latent variable expectations are estimated assuming the model parameters $\mathbf{\Theta}$ are constant.

"E"-step :

In the "E"-step the expectation of latent variables $\mathbf{z}$ are estimated based on $p(\mathbf{z}_n|\mathbf{x}_n,\mathbf{\Theta})$ for maximization, as proved in section 2.3.3. The 1st and 2nd order expectations of the latent variables $\mathbf{z}$ used in (2.63) and (2.67) are exactly the expected mean and covariance of $p(\mathbf{z}_n|\mathbf{x}_n,\mathbf{\Theta})$. To estimate these conditional expectations, we express the combined joint distribution of the data and the factors as

$$p\left(\begin{bmatrix}\mathbf{x}\\ \mathbf{z}\end{bmatrix}\right) = \mathcal{N}\left(\begin{bmatrix}\mathbf{0}\\ \mathbf{0}\end{bmatrix}, \begin{bmatrix}\mathbf{\Lambda}\mathbf{\Lambda}^T + \Psi & \mathbf{\Lambda}\\ \mathbf{\Lambda}^T & \mathcal{I}\end{bmatrix}\right) \tag{2.68}$$

For a multivariate Gaussian distribution is can be shown [24] that

$$\mathcal{E}_{\mathbf{z}}\{\mathbf{z}|\mathbf{x}\} = \mu_{z|x} = \mu_z + \mathbf{\Sigma}_{zx}\mathbf{\Sigma}_x^{-1}(\mathbf{x}-\mu_x) \tag{2.69}$$

$$\mathcal{E}_{\mathbf{z}}\{\mathbf{z}\mathbf{z}^T|\mathbf{x}\} = \mathbf{\Sigma}_{z|x} = \mathbf{\Sigma}_z - \mathbf{\Sigma}_{zx}\mathbf{\Sigma}_x^{-1}\mathbf{\Sigma}_{xz} \tag{2.70}$$

This means we can express expected value of the factors $\mathbf{z}$ by

$$\mathcal{E}_{\mathbf{z}}\{\mathbf{z}|\mathbf{x}\} = \mathbf{\Lambda}^T(\mathbf{\Psi} + \mathbf{\Lambda}\mathbf{\Lambda}^T)^{-1}\mathbf{x} \tag{2.71}$$

$$= \beta\mathbf{x} \tag{2.72}$$

where $\beta$ is defined as $\beta = \mathbf{\Lambda}^T(\mathbf{\Psi} + \mathbf{\Lambda}\mathbf{\Lambda}^T)^{-1}$. As $\mathbf{\Psi}$ is diagonal we can use the Woodbury matrix identity [24] to invert $(\mathbf{\Psi} + \mathbf{\Lambda}\mathbf{\Lambda}^T)^{-1}$ efficiently by

$$(\mathbf{\Psi} + \mathbf{\Lambda}\mathbf{\Lambda}^T)^{-1} = \mathbf{\Psi}^{-1} - \mathbf{\Psi}^{-1}\mathbf{\Lambda}(\mathcal{I} + \mathbf{\Lambda}^T\mathbf{\Psi}^{-1}\mathbf{\Lambda})^{-1}\mathbf{\Lambda}^T\mathbf{\Psi}^{-1} \tag{2.73}$$

where $\mathcal{I}$ is the $d' \times d'$ identity matrix. The second moment of the factors can be found by

$$\begin{aligned} \mathcal{E}_{\mathbf{z}}\{\mathbf{z}\mathbf{z}^T|\mathbf{x}\} &= \mathrm{var}(\mathbf{z}|\mathbf{x}) + \mathcal{E}_{\mathbf{z}}\{\mathbf{z}|\mathbf{x}\}\mathcal{E}_{\mathbf{z}}\{\mathbf{z}|\mathbf{x}\} & (2.74)\\ &= \mathcal{I} + \mathbf{\Lambda}^T(\mathbf{\Psi} + \mathbf{\Lambda}\mathbf{\Lambda}^T)^{-1}\mathbf{\Lambda} + (\beta\mathbf{x})(\beta\mathbf{x})^T & (2.75)\\ &= \mathcal{I} - \beta\mathbf{\Lambda} + \beta\mathbf{x}\mathbf{x}^T\beta^T & (2.76) \end{aligned}$$

This gives a measure of uncertainty of the factors $\mathbf{z}$, a quantity with no analogue in PCA. Having introduced the linear Factor Analyzer model we expand it to a mixture model using FA.

### 2.5.2 Mixture of Factor Analyzers

Using the FA as the kernel function we can model an observed dataset $\mathbf{X} = \{\mathbf{x}_n\}_{n=1}^N$ by a mixture of $K$ Factor Analyzers. The mixture model can be expressed based on the general form given in (2.29) (omitting samples indices) as

$$p(\mathbf{x}) = \sum_{k=1}^{K} \alpha_k \cdot \int p(\mathbf{x}|\mathbf{z},k)p(\mathbf{z}|k)\ \mathrm{d}\mathbf{z} \tag{2.77}$$

where $\alpha_k = p(k)$ is the component prior summing to 1, i.e. $\sum_k \alpha_k = 1$ and the factors $\mathbf{z}$ are assumed to be $\mathcal{N}(0,\mathcal{I})$ distributed as in regular factor analysis, i.e. $p(\mathbf{z}|k) = p(\mathbf{z}) = \mathcal{N}(0,\mathcal{I})$. Comparing with the general form in (2.29) $p(\mathbf{x}|k)$ can in this case be found by marginalizing over all latent variables $\mathbf{z}$, i.e. $p(\mathbf{x}|k) = \int p(\mathbf{x}|\mathbf{z},k)p(\mathbf{z},k)\ \mathrm{d}\mathbf{z}$. In factor analysis the mean of the observed data $\mathbf{x}$ was assumed zero, but for the mixture model we assign a $d$ dimensional mean vector to each FA denoted $\mu_k$. The conditional distribution of $\mathbf{x}$ can then be written as

$$p(\mathbf{x}|\mathbf{z},k) = \mathcal{N}(\mu_k + \mathbf{\Lambda}_k\mathbf{z}, \mathbf{\Psi}) \tag{2.78}$$

For the MFA the model parameters to estimate therefore become $\mathbf{\Theta} = \big\{\{\alpha_k, \mu_k, \mathbf{\Lambda}_k\}_{k=1}^K, \mathbf{\Psi}\big\}$. If we re-introduce the binary variable $\mathbf{h}_n = \{h_{k,n}\}_{k=1}^K$ as a mixture indicator similar for the MoG model in (2.31) and identify the factors $\mathbf{z}$ and the mixture weight $\mathbf{h}$ as the latent variables for this model we can expand or modify the "EM"-algorithm from FA to estimate the model parameters. From the conditional distribution (2.78) and (2.56) we can redefine the expected log-likelihood $Q(\mathbf{\Theta})$ from (2.61) to

$$Q(\mathbf{\Theta}) = \mathcal{E}\Bigg\{ \ln \prod_k \prod_n \Bigg[\frac{1}{\sqrt{(2\pi)^d|\mathbf{\Psi}|}} \exp\Big(-\frac{1}{2}(\mathbf{x}_n - \mu_k - \mathbf{\Lambda}_k\mathbf{z})^T\mathbf{\Psi}^{-1}(\mathbf{x}_n - \mu_k - \mathbf{\Lambda}_k\mathbf{z})\Big)\Bigg]^{h_{k,n}}\Bigg\} \tag{2.79}$$

where $h_{k,n}$ serves to only activate the mixture component responsible for generating $\mathbf{x}_n$, as seen before. To jointly estimate the mean $\mu_k$ and the factor loading matrix $\mathbf{\Lambda}_k$, we define an augmented column vector of factors and factor loading matrices

$$\tilde{\mathbf{\Lambda}}_k = [\mathbf{\Lambda}_k\ \mu_k] \qquad \wedge \qquad \tilde{\mathbf{z}} = \begin{bmatrix} \mathbf{z} \\ 1 \end{bmatrix} \tag{2.80}$$

The expected log-likelihood in (2.79) can then be rewritten as

$$Q(\boldsymbol{\Theta}) = \mathcal{E}\Bigg\{ \ln \prod_k \prod_n \Bigg[ \frac{1}{\sqrt{(2\pi)^d |\boldsymbol{\Psi}|}} \exp\Big( -\frac{1}{2}(\mathbf{x}_n - \tilde{\boldsymbol{\Lambda}}_n \tilde{\mathbf{z}})^T \boldsymbol{\Psi}^{-1} (\mathbf{x}_n - \tilde{\boldsymbol{\Lambda}}_n \tilde{\mathbf{z}}) \Big) \Bigg]^{h_{k,n}} \Bigg\} \tag{2.81}$$

$$= c - \frac{N}{2} \ln |\boldsymbol{\Psi}| - \sum_{k,n} \frac{1}{2} w_{k,n} \mathbf{x}_n^T \boldsymbol{\Psi}^{-1} \mathbf{x}_n - w_{k,n} \mathbf{x}_n^T \boldsymbol{\Psi}^{-1} \tilde{\boldsymbol{\Lambda}}_n \mathcal{E}_{\mathbf{z}}\{\tilde{\mathbf{z}} | \mathbf{x}_n, k\}$$
$$+ \frac{1}{2} w_{k,n} \mathrm{tr}\big( \tilde{\boldsymbol{\Lambda}}_k^T \boldsymbol{\Psi}^{-1} \tilde{\boldsymbol{\Lambda}}_k \mathcal{E}_{\mathbf{z}}\{\tilde{\mathbf{z}}\tilde{\mathbf{z}}^T | \mathbf{x}_n, k\} \big) \tag{2.82}$$

where $c$ is a constant independent of the model parameters and expectation of the mixture indicators $w_{k,n} = \mathcal{E}\{h_{k,n}\}$ are proportional to the joint distribution of $\mathbf{x}_n$ and $h_{k,n}$, i.e. using Bayes

$$w_{k,n} = \mathcal{E}\{h_{k,n}\} \propto p(\mathbf{x}_n, h_{k,n}) = p(\mathbf{x}_n | h_{k,n}) p(h_{k,n}) \tag{2.83}$$
$$= \alpha_k \, \mathcal{N}\{\mathbf{x}_n - \mu_k, \boldsymbol{\Lambda}_k \boldsymbol{\Lambda}_k^T + \boldsymbol{\Psi}\} \tag{2.84}$$

The expression in (2.82) is used for deriving the model parameters in the "M"-step, shown next

"M"-step :

The model parameters $\boldsymbol{\Theta} = \big\{ \{\alpha_k, \mu_k, \boldsymbol{\Lambda}_k\}_{k=1}^K, \boldsymbol{\Psi} \big\}$ are estimated by setting the derivative of $Q(\boldsymbol{\Theta})$ to zero for the respective parameters. Initially we derive the update equation for $\tilde{\boldsymbol{\Lambda}}_k$ by

$$\frac{\partial Q(\boldsymbol{\Theta})}{\partial \tilde{\boldsymbol{\Lambda}}_k^{(m)}} = -\sum_{n=1}^{N} w_{k,n} \boldsymbol{\Psi}^{-1} \mathbf{x}_n \mathcal{E}_{\mathbf{z}}\{\tilde{\mathbf{z}} | \mathbf{x}_n, k\}^T - w_{k,n} \boldsymbol{\Psi}^{-1} \tilde{\boldsymbol{\Lambda}}_k^{(m+1)} \mathcal{E}_{\mathbf{z}}\{\tilde{\mathbf{z}}\tilde{\mathbf{z}}^T | \mathbf{x}_n, k\} = 0 \tag{2.85}$$

If we isolate for $\tilde{\boldsymbol{\Lambda}}_k^{(m+1)}$, we get the update equation for the factor loading matrices

$$\tilde{\boldsymbol{\Lambda}}_k^{(m+1)} = \big[ \boldsymbol{\Lambda}_k^{(m+1)} \;\; \mu_k^{(m+1)} \big] = \Bigg( \sum_{n=1}^{N} w_{k,n} \mathbf{x}_n \mathcal{E}_{\mathbf{z}}\{\tilde{\mathbf{z}} | \mathbf{x}_n\}^T \Bigg) \Bigg( \sum_{n=1}^{N} w_{k,n} \mathcal{E}_{\mathbf{z}}\{\tilde{\mathbf{z}}\tilde{\mathbf{z}}^T | \mathbf{x}_n\} \Bigg)^{-1} \tag{2.86}$$

where the augmented expectations of the factors $\tilde{\mathbf{z}}$ are

$$\mathcal{E}_{\mathbf{z}}\{\tilde{\mathbf{z}} | \mathbf{x}_n\} = \begin{bmatrix} \mathcal{E}_{\mathbf{z}}\{\mathbf{z} | \mathbf{x}_n\} \\ 1 \end{bmatrix} \qquad \wedge \qquad \mathcal{E}_{\mathbf{z}}\{\tilde{\mathbf{z}}\tilde{\mathbf{z}}^T | \mathbf{x}_n\} = \begin{bmatrix} \mathcal{E}_{\mathbf{z}}\{\mathbf{z}\mathbf{z}^T | \mathbf{x}_n\} & \mathcal{E}_{\mathbf{z}}\{\mathbf{z} | \mathbf{x}_n\} \\ \mathcal{E}_{\mathbf{z}}\{\mathbf{z} | \mathbf{x}_n\} & 1 \end{bmatrix} \tag{2.87}$$

The update equation for $\boldsymbol{\Psi}$ we derive through its inverse $\boldsymbol{\Psi}^{-1}$ as we did for regular FA earlier. The derivate from (2.82) becomes

$$\frac{\partial Q(\boldsymbol{\Theta})}{\partial \boldsymbol{\Psi}^{-1(m)}} = \frac{N}{2} \boldsymbol{\Psi}^{(m+1)} - \sum_{k,n} \Bigg( \frac{1}{2} w_{k,n} \mathbf{x}_n \mathbf{x}_n^T - w_{k,n} \tilde{\boldsymbol{\Lambda}}_{\mathbf{k}}^{(m+1)} \mathcal{E}_{\mathbf{z}}\{\tilde{\mathbf{z}} | \mathbf{x}_n, k\} \mathbf{x}_n^T + \frac{1}{2} w_{k,n} \tilde{\boldsymbol{\Lambda}}_{\mathbf{k}}^{(m+1)} \mathcal{E}_{\mathbf{z}}\{\tilde{\mathbf{z}}\tilde{\mathbf{z}}^T | \mathbf{x}_n, k\} \tilde{\boldsymbol{\Lambda}}_{\mathbf{k}}^{T(m+1)} \Bigg) = 0 \tag{2.88}$$

By substituting $\tilde{\boldsymbol{\Lambda}}_{\mathbf{k}}^{(m+1)}$ with (2.86), we conduct the same manipulation as in (2.65) - (2.66) and by forcing the diagonal constraint on $\boldsymbol{\Psi}$ we obtain

$$\boldsymbol{\Psi}^{(m+1)} = \frac{1}{N} \mathrm{diag} \Bigg\{ \sum_{n,k} w_{k,n} \Big( \mathbf{x}_n \tilde{\boldsymbol{\Lambda}}_k^{(m+1)} \mathcal{E}_{\mathbf{z}}\{\tilde{\mathbf{z}} | \mathbf{x}_n, k\} \Big) \, \mathbf{x}_n^T \Bigg\} \tag{2.89}$$

The component priors $\alpha_k$ can be computed using the same derivation as for eq. (2.46), hence

$$\alpha_k = \frac{1}{N} \sum_{n=1}^{N} w_{k,n} \tag{2.90}$$

All the update equations for the MFA model in (2.86), (2.89) and (2.90) can easily be compared to the equivalent expression derived for the MoG model earlier with the additional weighting factor $w_{k,n}$. This is the same intuitive result as we derived for the MoG model in section 2.4.

"E"-step :

In the "E"-step we need to estimate the expectation of all the latent variables $\mathbf{z}$ and $h$ and their interactions found in (2.86), (2.89) and (2.90) under the assumption that the model parameters $\boldsymbol{\Theta}$ are constant. With reference to the definition of $\mathcal{E}_{\mathbf{z}}\{\mathbf{z}|\mathbf{x}\}$ in (2.72) and $\mathcal{E}_{\mathbf{z}}\{\mathbf{z}\mathbf{z}^T|\mathbf{x}\}$ in (2.76), we can express the 1st and 2nd order conditional expectation of $\mathbf{z}$ as

$$\mathcal{E}_{\mathbf{z}}\{\mathbf{z}|\mathbf{x},k\} = \beta_k(\mathbf{x}_n - \mu_{\mathbf{k}}) \qquad \wedge \qquad \mathcal{E}_{\mathbf{z}}\{\mathbf{z}\mathbf{z}^T|\mathbf{x},k\} = -\beta_k\boldsymbol{\Lambda}_k + \beta_k(\mathbf{x}_n - \mu_k)(\mathbf{x}_n - \mu_k)^T\beta_k^T \tag{2.91}$$

The expectation of the latent mixture indicator $w_{k,n}$ was derived earlier in eq. (2.50), where we use $p(\mathbf{x}_n|k) = \mathcal{N}\{\mathbf{x}_n - \mu_k, \boldsymbol{\Lambda}_k\boldsymbol{\Lambda}_k^T + \boldsymbol{\Psi}\}$ for the MFA model.

The "EM"-algorithm presented for the MFA model is available as a toolbox in MATLAB [5] and can be found on the associated DVD.

Having presented the MFA model is can easily be seen that the covariance information is modelled with more parameters than the MoG model. This leads to a threshold of number of parameters above which MFA does not contribute further to the modelling of covariance as then the effective number of parameters is less than those apparent from eq. (2.53). In other words the MFA cannot model the covariance better with more parameters up to this threshold.

## 2.6 Density Estimation of Cognitive Data

Up this point we have presented and discussed density estimating algorithms and shown techniques to control model complexity. In this section we will shortly discuss the influence and limitations using density estimation in the context of cognitive data.

As discussed earlier we can divide the dataspace of the sources into a valid and an invalid part to help identify the multidimensional space to model. The objective of our mixture model is thus to be able to best model or capture the generalized boundaries between these two spaces such that new valid samples can be extracted. Figure 2.5 shows the valid/invalid split-up for three different modelling cases.

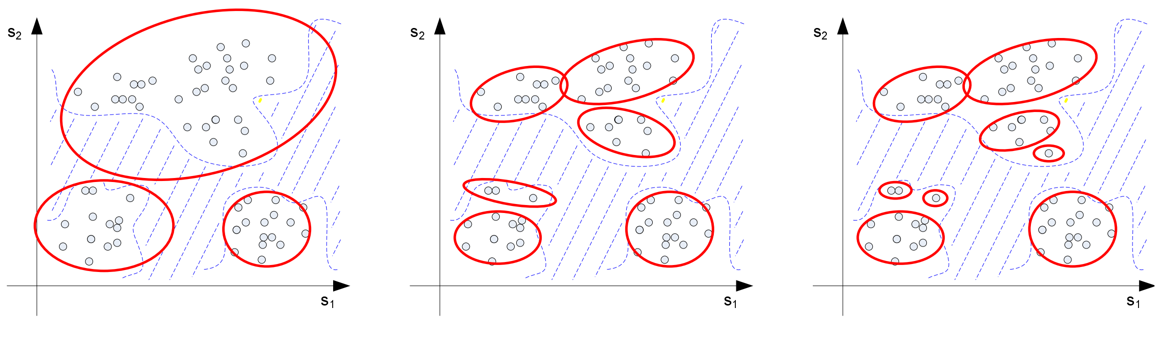

*Figure 2.5:* Illustration of invalid dataspace (blue dashed area) with 3 cases of modelling.

The left illustration in figure 2.5 shows a mixture model, which suffers from a bias (underfit) and thus also models invalid dataspace. Increasing the model complexity gives a better fit and limits the generation of invalid data, as shown in the middle illustration. If the complexity is further increased as shown in right illustration, the model tends to overfit and adapt to specific datapoints. This of course does not yield invalid data, but limits the flexibility of the model in generalizing to new samples. Thus the choice of model complexity becomes a tradeoff between the share of invalid data modelled and model overfitting.

[5] Available from the website of Prof. Zoubin Ghahramani at http://learning.eng.cam.ac.uk/zoubin/software.html

# 3. Mixture Model Simulations

In section 2 we introduced the class of mixture models based on the linear model $\mathbf{x} = \mathbf{A}\mathbf{s}$ as the generative model, where $\mathbf{x}$ is the observed data. For the actual decomposition of $\mathbf{x}$ we aim to extract features $\mathbf{A} = \{\mathbf{a}_r\}_{r=1}^{d}$, which are similar to cognitive components. Based on these cognitive features, we present different generative models formed from cluster analysis, mixture of Gaussians and mixture of factor analyzers and evaluate their performance in terms generation of new data. The MATLAB code used for the simulations can be found in appendix B.2 and on the DVD.

For the simulations we use a subset of the *MNIST* dataset, which will be described next in the first subsection prior to the description of the models.

## 3.1 The MNIST Dataset

The dataset we use in our simulations is the MNIST (*Modified National Institute of Standards and Technology*) database of handwritten digits from 0 to 9[6]. Originally the US Post Office wanted to automate the process of sorting letters based on their zip-codes. This lead to the digitization of app. 70000 handwritten zipcode digits from American letters and eventually forming the MNIST database.

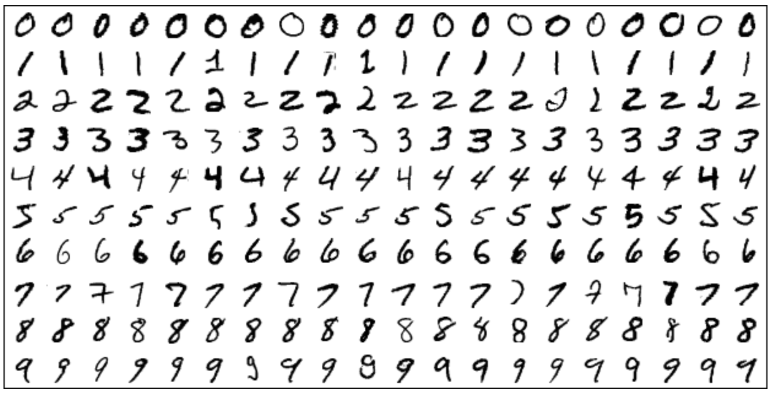

*Figure 3.1:* Original MNIST dataset all digit classes (only 20 samples shown).

In the MNIST dataset each digit sample is represented as a $28 \times 28$ pixel B/W image forming a $M = 28 \times 28 = 784$ dimensional vector. Figure 3.1 shows 20 examples from each digit class 0 to 9.

This leads to a vector-space representation of a total of $70000$ column vectors $\mathbf{x}_n$ in a large data matrix $\mathbf{X} = \{\mathbf{x}_n\}_{n=1}^{N}$, where $N = 70000$ and thus $\mathbf{X}$ becomes a $784 \times 70000$ data matrix for all digits. For each digit class the data can further be divided into a large training set and a smaller testset given in table 3.1 below.

| Digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Trainingset | 5923 | 6742 | 5958 | 6131 | 5842 | 5241 | 5918 | 6265 | 5851 | 5949 | 60000 |
| Testset | 980 | 1135 | 1032 | 1010 | 982 | 892 | 958 | 1028 | 974 | 1009 | 10000 |

*Table 3.1:* Subgroups of the MNIST dataset.

In our simulations it is important to have a dataset, which represents as many properties from the MNIST dataset as possible. We therefore choose to use digit 2 due to its good structural properties, as it includes both straight lines and curves. This means in evaluating the performance of any algorithm our dataset $\mathbf{X}$ will comprise only of the corresponding training set for the 2 digit, i.e. the dataset reduces to $\mathbf{X} = 784 \times 5958$ matrix.

[6]The MNIST database is available for download from http://yann.lecun.com/exdb/mnist/.

Pre-processing

All digits in the MNIST dataset are originally represented with integer pixel values between $[0 : 255]$. As a pre-processing step all components in the vectors (pixels in the digits) $\mathbf{x} = \{x_1, x_2, \ldots x_M\}$ are normalized to unity dynamic range, i.e. $\mathbf{x}_{i,n} \leftarrow \frac{1}{255}\mathbf{x}_{i,n}$. This means each pre-processed data sample has an average energy of appr. $E_{avg} = 89$.

Post-processing and Visualization

Throughout our analysis the generated or produced data $\mathbf{X}$ can have various formats in term of dynamic range and dimensionality. This requires some post-processing in order to align the formats for data viewing. When displaying any $784$ dimensional digit vector $\mathbf{x}$ the individual image patches are scaled to $[0 : 1]$ to maximize contrast and inverted to give a white background. Refer to appendix B.1 for the `MATLAB` code.

Some of the models employed include negative values in their dynamic range, e.g. the MoG model, as we shall see later. Since image data is strictly non-negative any negative generated values are therefore considered illegal and are truncated. It is important to note that this processing only applies when viewing data and it not part of the analysis itself.

Visual Evaluation

In generating new digits from any of our models it becomes hard to define a quantity, which express the quality of a generated digit, i.e. does a generated digit resemble anything handwritten, including the BIC introduced in section 2.3.2. This means we can only evaluate the quality of any generated digit visually.

Without such a quality measure it also becomes hard find optimal parameters for the respective generative model in use. This will also have to be evaluated visually and by trial'n'error. In the following sections the result from the individual models will be presented and evaluated visually without any measurable quality measure.

## 3.2 Extracting Features

For extracting linear features from the observed data $\mathbf{x}$ numerous methods with different constraints exist, such as PPCA [29], ICA [16] or NMF. In the context with image data holding only non-negative pixels it seems reasonable to apply *Non-Negative Matrix Factorization* (NMF) for a non-negative parts-based cognitive feature extraction of $\mathbf{x}$.

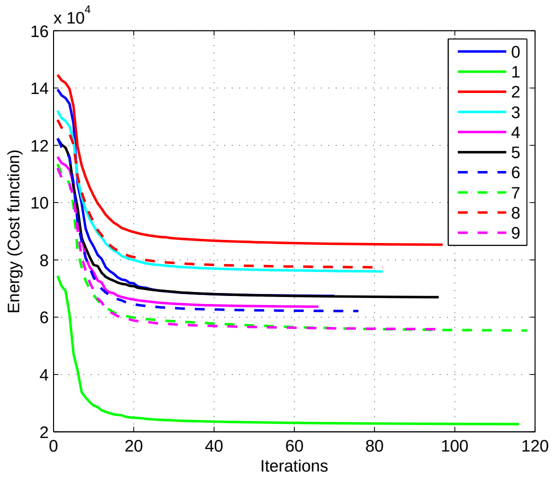

*Figure 3.2:* Convergence of NMF cost-function for digit 2 for $d = 10$.

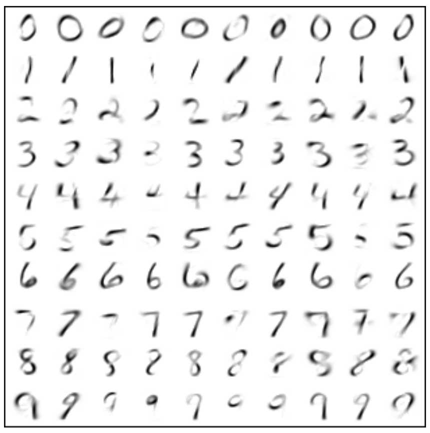

*Figure 3.3:* Separate NMF features for all digit classes, i.e. column vectors of $\mathbf{A}$, for $d = 10$.

As part of extracting features with NMF, the number of features $d$ must be specified as a model parameter (refer to section 2.1). This means if $d$ is too small we might achieve features, which are similar to the original observed sample and thus fail to capture underlying components. In contrast if $d$ is too high, we risk decomposing into parts which are close to pixel-level structures. In order to find an optimal number of features, the NMF algorithm implemented as `SNMF2D` [7] (refer to the DVD for `MATLAB` code) is applied with different values of $d = \{10, 20, 30, 40, 50, 100, 200\}$. Figure 3.2 shows the successful convergence of the energy-based cost-function defined in (2.3) during the NMF decomposition of all digits separately for $d = 10$.

To illustrate the effect of different amount of features, figure 3.3 depicts the non-negative features for all digits separately for $d = 10$. These features clearly resemble their respective original digit and suggests that the number of features $d = 10$ is too low, as mentioned earlier. In addition each row in figure 3.3 shows a subset of the features extracted for different values of $d = \{20, 30, 40, 50, 100, 200\}$. The figure clearly shows how the features becomes more sparse as $d$ increases. Judging from a visual subjective point, the size of $d = 30$ (2nd row) represents a feature set, which resemble cognitive components. Thus choosing $d = 30$ for our initial simulations therefore seems justified.

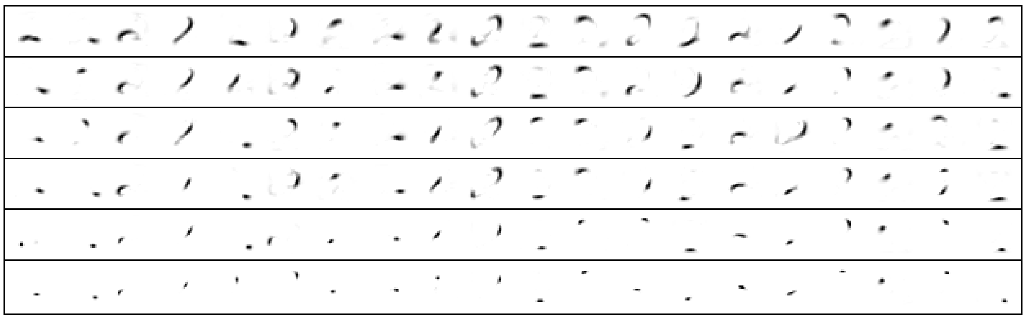

*Figure 3.4:* Extracted NMF features for the 2 digit, where each row from the top represents different values $d = \{20, 30, 40, 50, 100, 200\}$ respectively.

In addition the decomposition of $\mathbf{x}$ introduces the error $E_{LS}$ (2.3), which is inverse proportional to $d$ and can be seen on figure 3.6. To illustrate the effect each row in figure 3.5 shows random reconstructed digits for all classes for different values of $d$ respectively.

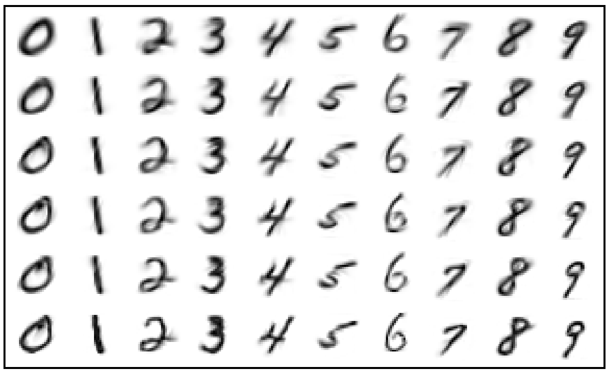

*Figure 3.5:* Random reconstructed digits, where each row represents $d = \{10, 20, 30, 40, 50, 100\}$ respectively.

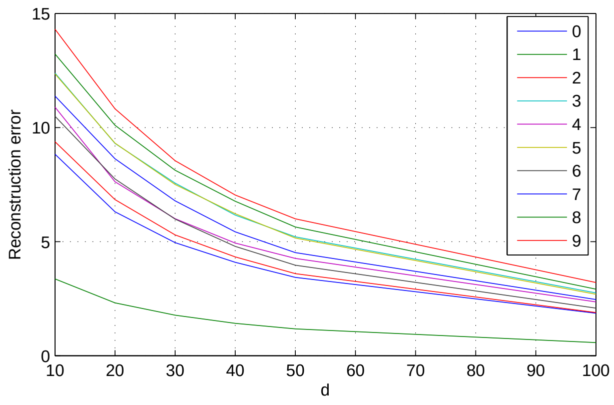

*Figure 3.6:* Reconstruction error for $d = \{10, 20, 30, 40, 50, 100\}$.

This figure illustrates how digits for low values of $d$ are reconstructed fairly good, where all digits can be identified clearly. Increasing $d$ only adds visual features with low dynamic range or variance. The small visible error for the lowest values of $d$ is as mentioned not enough to misclassify any digits. In particular digits 2 and 3 suffer from the largest error-level (still sufficiently small though). This again suggests that the 2 digit is among the hardest to generate sufficiently and thus choosing the 2 digit for our analysis seems reasonable.

In the following sections we present different clustering algorithms using the linear NMF feature codebook $\mathbf{A}$ and encoding vectors $\mathbf{s}$ to build a linear generative model.

[7] Available at http://www.imm.dtu.dk/pubdb/views/edoc_download.php/4521/zip/imm4521.zip.

## 3.3 Codebook Clustering by K-Means

In this section we introduce a generative model, where the modelling of the sources $\mathbf{s}$ is not in focus, but where a digit $\mathbf{x}$ is generated solely by selecting elements from the codebook $\mathbf{A} = \{\mathbf{a}_r\}_{r=1}^{d}$. This is can of course be seen as a different way of generating a valid source $\mathbf{s}$, but in this approach no information from the distribution of the sources is used. The basic concept is to exclude similar elements from being used in the generation of a digit $\mathbf{x}$ to avoid overlap and thereby only select different segments. If we assume the codebook elements $\mathbf{a}_r$ can be grouped into $K$ categories denoted $\phi_k$, such that $\mathbf{A} = \{\phi_k\}_{k=1}^{K}$, the generation of a digit $\mathbf{x}$ can be expressed as

$$\mathbf{x} = \sum_{k=1}^{K} \mathbf{a}_k \tag{3.1}$$

where we only select $K$ segments from the entire codebook $\mathbf{A}$. To group similar codebook elements $\mathbf{a}_r$ the K-Means algorithm described in section 2.2 is applied. Afterwards each of the elements $\mathbf{a}_r$ are associated as a member of one of the groups $\phi_k$, e.g. $\phi_2 = \{\mathbf{a}_3, \mathbf{a}_6, \mathbf{a}_{10}\}$. A digit $\mathbf{x}$ is then generated by extracting exactly one member $\mathbf{a}_k$ from each group $\phi_k$ with similar elements with equal probability and adding these contributions as defined in (3.1). The pseudocode for the algorithm is listed in table 3.2 (`MATLAB` implementation is given in appendix B.2 )

```
1. Group the codebook elements $\mathbf{a}_r$, where $r = 1, 2, \ldots, d$ into $K$ clusters denoted $\phi_k$, where $k = 1, 2, \ldots, K$ using K-Means, refer to section 2.2.
2. For all clusters $\phi_k$ extract a single segment $\mathbf{a}_k$ with equal probability, i.e. $p_k(\mathbf{a}_k) = \frac{1}{l_k}$, where $l_k$ is the number of members in cluster $\phi_k$.
3. Generate a digit $\mathbf{x}$ by adding all $K$ extracted segments $\mathbf{a}_k$ as defined in (3.1).
4. Repeat steps 2 - 3 to generate additional digits $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots\}$
```

*Table 3.2:* Pseudocode for the K-Means algorithm

Each codebook element $\mathbf{a}_r$ is used discretely in the generation of a digit with no linear weighting. This sets one of the greatest limitation of this approach, that it can only generate a limited amount of digits, max. $\prod_{k=1}^{K} |\phi_k|$ restricted by the size of the codebook $d$ and the amount of clusters $K$.

In addition the number of clusters $K$ in this model is a sensitive parameter. The amount should represent the number of cognitive components for the particular digit, so that each cluster contains all overlapping similar segments. This is obviously the big challenge for the K-Means algorithm in this context. If $K$ is too high we risk splitting up the cognitive components into several clusters and thereby achieve a potential overlap in the generation of digits. In contrast if $K$ is too low we risk having non-similar segments in the same cluster and get potential holes or incoherency in the generated digits. The performance of the generation of digit can therefore be loosely predicted by analyzing the resulting clustering of the codebook elements.

With the algorithm described a set of simulations is conducted to evaluate its generative performance.

Results:

In the initial simulation, we select a small codebook $\mathbf{A}$ for the digit 2 with $d = 30$ elements shown in figure 3.7. Choosing $d = 30$ ensures a codebook where the individual segments can be considered similar to cognitive components.

For the 2 digits used in these simulations, we cluster the codebook $\mathbf{A}$ into $K = 5$ components using the K-Means algorithm with random datapoints as initial clusters, $\mu_{init}$. Figure 3.8 shows a scatterplot (1st and 2nd principal component, refer to appendix A.1) of the codebook datapoints with the initial cluster centers $\mu_{init}$ and final means $\mu_k$.

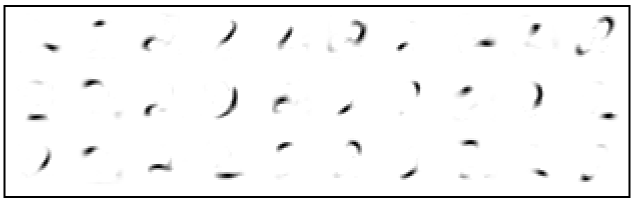

*Figure 3.7:* NMF features for $d = 30$.

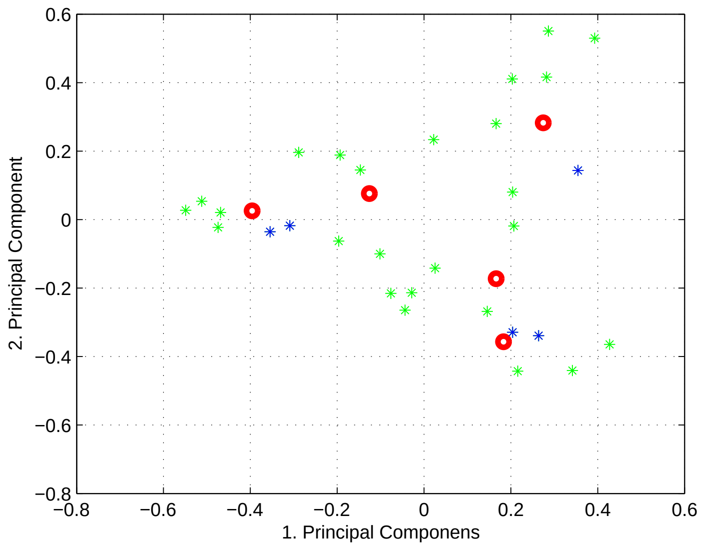

*Figure 3.8:* Scatterplot of codebook data (green dots) with initial cluster centers (blue dots) and final means (red dots), all projected onto the 1st and 2nd principal axis for $K = 5$.

The figure clearly shows how the decorrelated datapoints (green dots) are spread widely with no immediate group structure. In addition it is evident to see that all cluster centers $\mu_k$ were updated during the iterations. The final means (red dots) are illustrated as real image patches in figure 3.9.

Due to the relatively low amount of clusters $K$, figure 3.9 reveals how the cluster centers resemble half-complete 2 digits instead of cognitive segments. This means we risk achieving imperfect grouping of the codebook elements in $A$ and hence get a mixture of unrelated elements in the same group $\phi_k$, since unrelated segments might be associated with the same cluster. The resulting cluster associations $\phi_k$ are illustrated in figure 3.10.

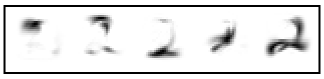

*Figure 3.9:* Cluster centers for $K = 5$

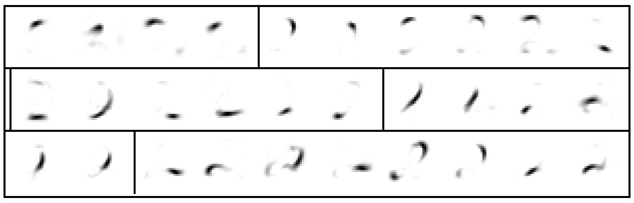

*Figure 3.10:* Sorted NMF features for $d = 30$ and $K = 5$ clusters.

This figure clearly shows how the K-Means algorithm has clustered similar segments fairly good. Still some groups contains few non-similar elements, which does not represent the same segment of the 2 digit, particularly the last group $\phi_5$. This can lead to either overlapping segments or incoherency (holes in the structure) of any generated digits as mentioned.

A set of digits was generated by repeating the algorithm defined in table 3.2 with different random initial clusters. Each row in figure 3.11 shows a set of 20 random generated digits from each run, where the last row is generated from the clustering shown in figures 3.9 and 3.10.

This reveals poorly generated 2 digits suffering from exactly overlapping and incoherent segments independent

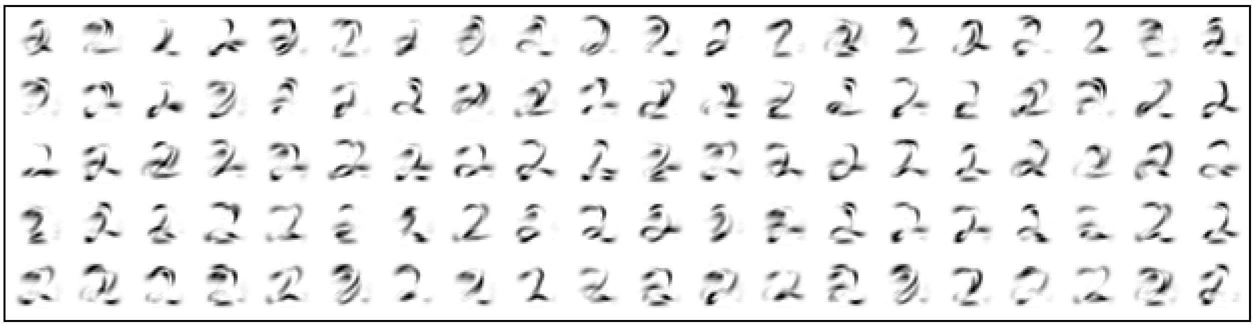

*Figure 3.11:* Random generated 2 digits for $d = 30$ features and $K = 5$ clusters, where each row has a different set of random initial clusters.

of the initial clusters. It is still clear to identify the 2 digits, but they do certainly not represent any handwritten 2 digit. This is due to the poor clustering of the K-Means algorithm resulting in the sorted codebook in figure 3.10, where the groups contain non-similar segments, as mentioned.

To improve the clustering the number of clusters is increased to $K = 8$ for the same codebook size $d = 30$ and thereby allow greater freedom to group the codebook elements $\mathbf{a}_r$. Figure 3.12 illustrates the same type of scatterplot as before.

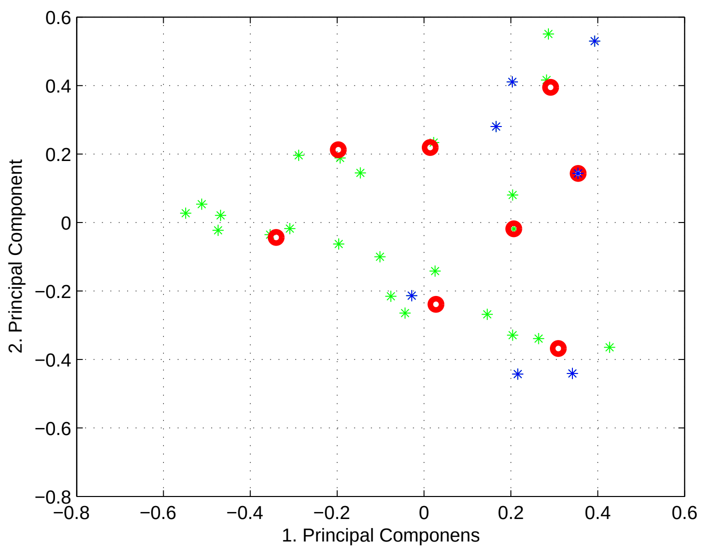

*Figure 3.12:* Scatterplot of codebook data (green dots) with initial cluster centers (blue dots) and final means (red dots), all projected onto the 1st and 2nd principal axis for $K = 8$.

From the figure a single cluster can be identified, which has failed to receive an update. This could suggest that there is a cluster with only one member associated, i.e. $|\phi_k| = 1$. This will be more evident later. Figure 3.13 shows the corresponding image patches for the cluster means $\mu_k$.

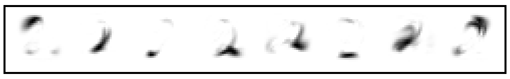

*Figure 3.13:* Cluster centers for $K = 8$

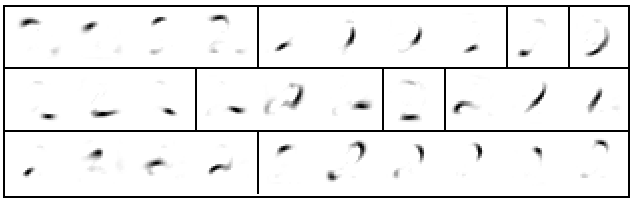

*Figure 3.14:* Sorted NMF features for $d = 30$ and $K = 8$ clusters.

The figure also clearly reveals the cluster means, which resemble half-complete 2 digits. This again indicates a risk of imperfect grouping as before as before for $K = 5$. Based on the cluster associations the grouping of the codebook elements $\mathbf{a}_r$ is illustrated in figure 3.14.

Initially the grouping of the codebook elements seems successful, but also reveals two clusters $3$ and $6$, which only has a single member associated, i.e. $|\phi_3| = 1$ and $|\phi_6| = 1$. This is very dependent on the initialization of the cluster centers $\mu_{init}$ and can be seen as an indication of too many clusters. As a direct consequence these two segments will always be used on the generation of all digits, which is both a limitation in the flexibility of the model and a risk for overlapping segments.

Using $K = 8$ clusters a set of simulations were again conducted with different initial clusters. Each row in figure 3.15 illustrates 20 generated digit from each run, where the last row is generated from the clustering shown in figure 3.13 and 3.14. From figure 3.15 is it clear to see how the generated digits still suffer from incoherency and overlapping segments. There are no clear generated digits, which can be characterized as handwritten and comparing with $K = 5$ it can be argued that the generation of digits has in fact been worse in terms of overlapping segments.

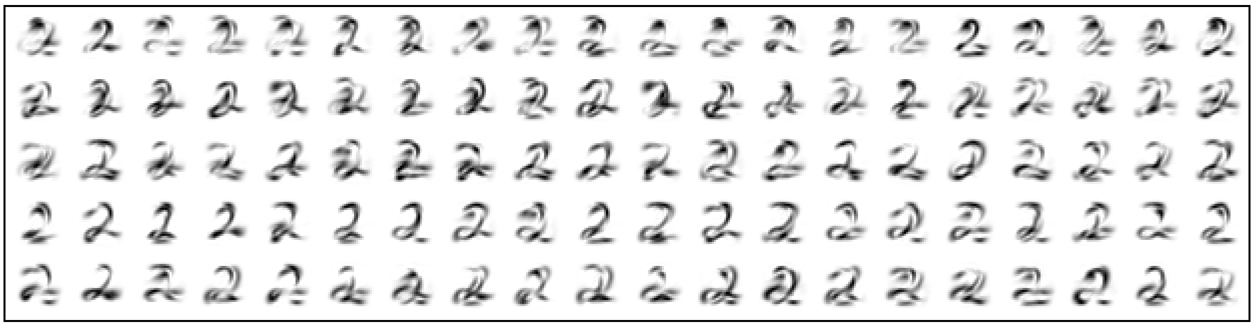

*Figure 3.15:* 100 generated 2 digits for $d = 30$ features and $K = 8$ clusters.

Increasing the number of clusters $K$ to achieve better performance therefore seems useless. Still to evaluate the performance of increasing amount of clusters each row in figure 3.16 illustrates 20 generated digits for $K = \{3, 5, 8, 10, 15, 20\}$ respectively.

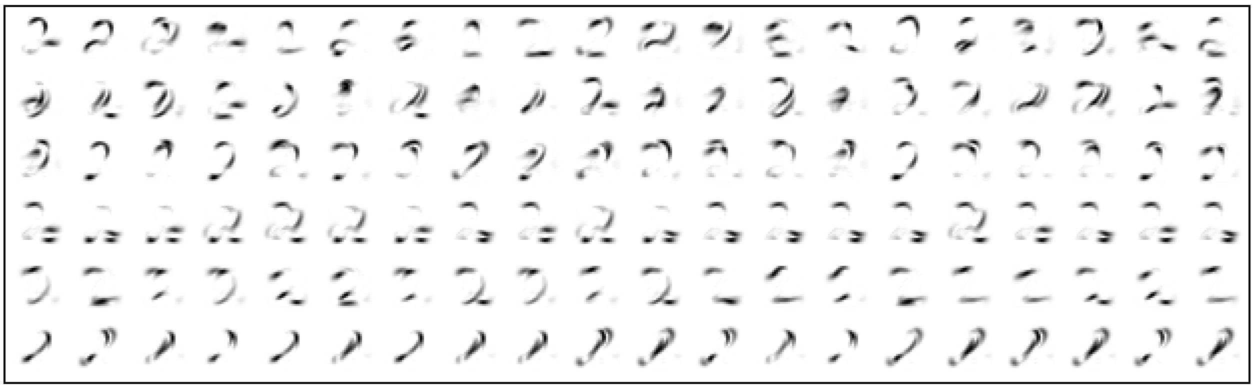

*Figure 3.16:* Generated 2 digits for $d = 30$ features and $K = \{3, 5, 8, 10, 15, 20\}$ clusters for each row respectively.

This figure clearly shows how the generated digits become more streamlined and similar as $K$ increases, in which case more and more clusters has only a single element constraining the flexibility in the generation of digits, as mentioned earlier.

Instead the size of the codebook $\mathbf{A}$ is expanded to $d = 100$ features. This leads to smaller segments $\mathbf{a}_r$ and hence additional number of cluster $K$ are required in order to avoid incoherency in the later generation of digits. The amount of clusters is therefore set to $K = 10$. The codebook $\mathbf{A}$ for with $100$ elements is shown in figure 3.17 as a reference.

After a few iterations, figure 3.18 again shows the resulting scatterplot.

For the larger codebook $d = 100$ the decorrelated datapoints $\mathbf{a}_r$ are wide spread still with a relatively strong concentration close to origo. From the figure it can be seen that every cluster has been updated during the iterations. The set of resulting cluster means $\mu_k$ are illustrated in figure 3.19.

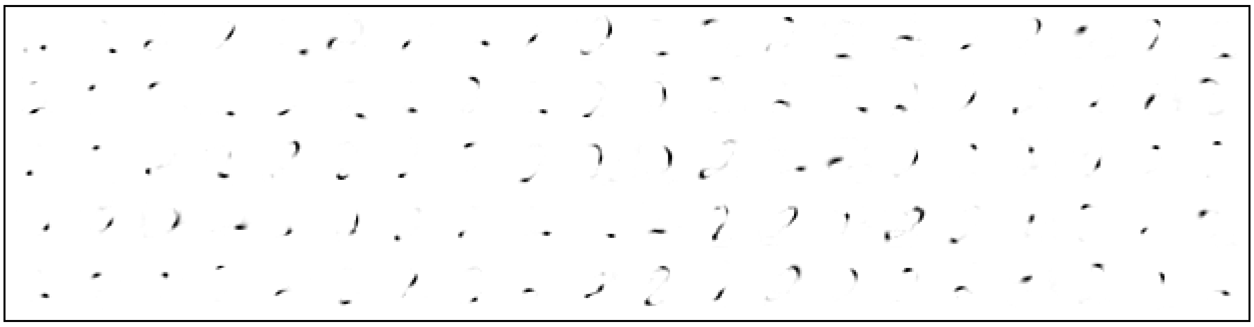

*Figure 3.17:* NMF features for $d = 100$.

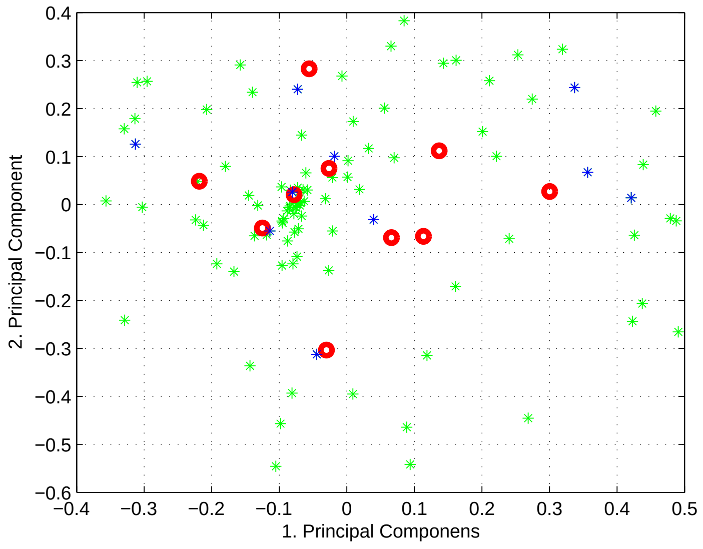

*Figure 3.18:* Scatterplot of codebook data (green dots) with initial cluster centers (blue dots) and final means (red dots), all projected onto the 1st and 2nd principal axis for $K = 10$.

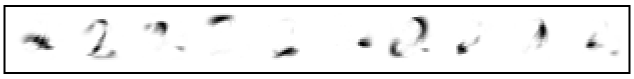

*Figure 3.19:* Cluster centers for $K = 5$.

These cluster centers also resemble half-complete digits each having specific structural features of the 2 digits. As before for $d = 30$ this gives potential imperfect grouping of the codebook elements. The resulting cluster associations $\phi_k$ are illustrated in figure 3.20.

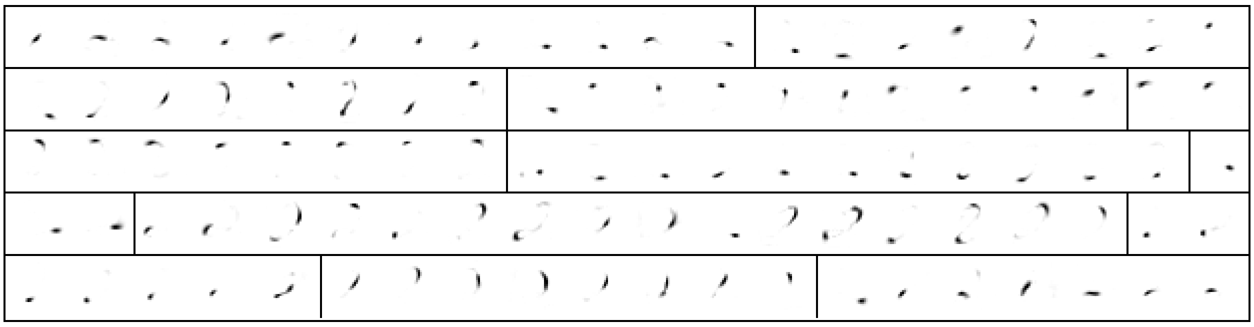

*Figure 3.20:* Sorted NMF features for $d = 100$ and $K = 5$ clusters.

The grouping shows how non-similar segment are associated to the same cluster as we have seen before. In this case for $d = 100$ it is even more evident compared to $d = 30$. Figure 3.21 shows 100 generated digit, where each row with 20 samples is generated from different initial clusters.

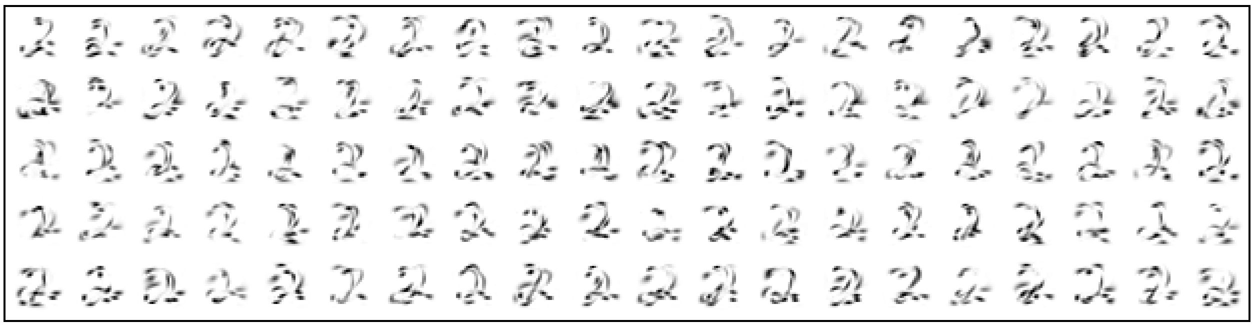

*Figure 3.21:* Random generated 2 digits for $d = 100$ features and $K = 10$ clusters, where each row has a different set of random initial clusters.

This illustration clearly shows both overlapping and incoherent digits, which hardly resemble anything handwritten. Based on the previous simulation result for $d = 30$ it is therefore not expected that an increase of the amount of clusters $K$ will improve the performance. To evaluate figure 3.22 shows 20 generated digit for different number of clusters $K = \{5, 10, 15, 20, 30, 40, 50\}$.

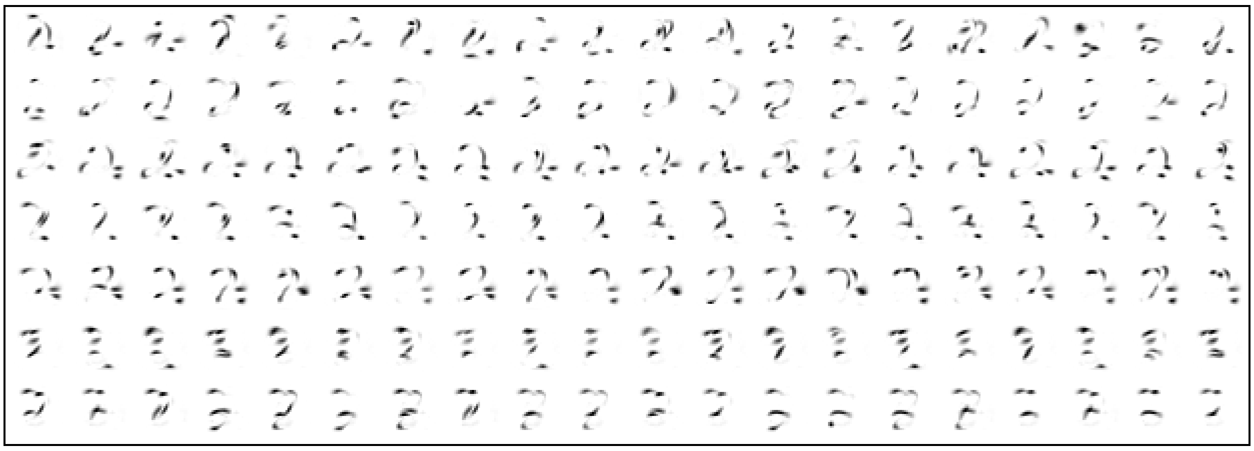

*Figure 3.22:* Generated 2 digits for $d = 100$ features and $K = \{5, 10, 15, 20, 30, 40, 50\}$ clusters for each row respectively.

This figure clearly shows how the different cluster sizes $K$ again has failed to generate valid digits. As $K$ increases the digits become more similar as the model becomes more restricted due to the increasing amount of clusters with single members as also seen for $d = 30$.

Summary

In general the algorithm presented in table 3.2 has not proven to be efficient in generating handwritten digits. The performance of the algorithm is very sensitive to the efficiency of the clustering of the K-Means method in terms of the amount of clusters $K$. As the segments are added discretely, i.e. with a weight of either $0$ or $1$, the flexibility of the algorithm is further restricted.

Having presented and analyzed the K-Means based algorithm, we proceed with a more sophisticated approach based on mixture of Gaussian distributions.

## 3.4 Direct Mixture of Gaussians Model

A natural advancement step in forming a generative model is to employ the mixture of Gaussians model as presented in section 2.4. By modelling the density of the data, we hope to better capture the underlying structure of the data and thus generate digits with improved quality. The MoG model can be used to model any probability density and the most obvious approach is to model the digits directly (hence the title of this section) and exclude any feature extraction as a pre-processing step. The MoG model can then be written as

$$p(\mathbf{x}) = \sum_{k=1}^{K} \alpha_k \cdot p(\mathbf{x}|\mathbf{\Theta}_k, k) \tag{3.2}$$

where $p(\mathbf{x}|\mathbf{\Theta}_k, k)$ is the multivariate Gaussian distribution defined in (2.30) and $\alpha_k$ is the component prior. To determine the mixture model parameters $\mathbf{\Theta}_k = \{\mu_k, \mathbf{\Sigma}_k, \alpha_k\}$ we use the EM-algorithm as defined in section 2.4. Using this approach we model the $768$ dimensional vector $\mathbf{x}$ directly with full covariance matrix, $\mathbf{\Sigma}^{full}$.

<u>Results:</u>

In evaluating the direct MoG model a set of different amount of components is used, $K = \{10, 20, 30, 50, 100, 200\}$ and to generate digit, we extract a random sample from the MoG as described in Appendix A.5. Afterwards the quality of the generated digits are evaluated visually.

For the simulations we used a log-likelihood tolerance of $\Delta\mathcal{L} = 1e - 3$ as a stopping criteria and a maximum of $IT_{max} = 500$ iterations. This initialization of the MoG model is used in all further simulations, unless otherwise noted.

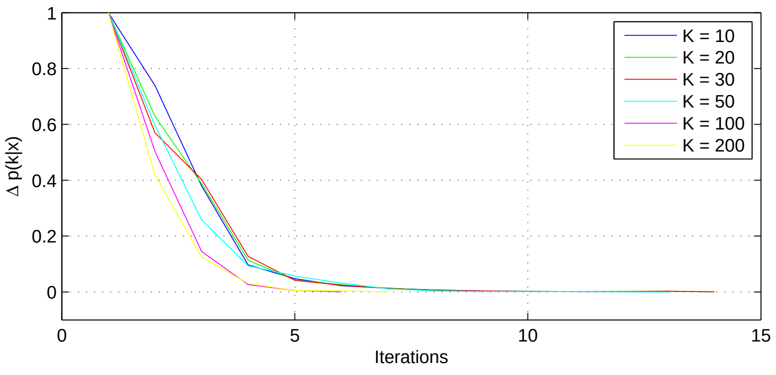

*Figure 3.23:* Convergence of the change in responsibility $w_k$ during iterations.

To evaluate the training of the MoG model, the evolution of the change of the responsibility $\Delta w_k$ (expectation of the hidden variables defined in section 2.4) is shown in figure 3.23. This figure shows how the "EM"-algorithm has converged within few iterations. For high-dimensional data $M = 784$ (relative to number of samples $N$) the covariance information for any direction is poorly determined due to the curse of dimensionality ($5958/784 = 7.6$ samples/dim.), which means the Gaussian mixtures can be shaped and distributed in an increasing number of ways. This makes it easy for the "EM"-algorithm to find a local maxima in log-likelihood space and leads to the few iterations.

The corresponding generated digits for $K = 10$ and $K = 30$ mixture components are shown below, refer to appendix A.5 to see how to generate samples from the trained mixture model.

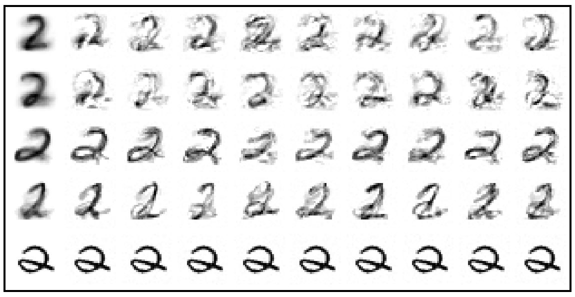

*Figure 3.24:* Direct modelling of the digits for $K = 10$ mixture components (mean vectors in 1st column).

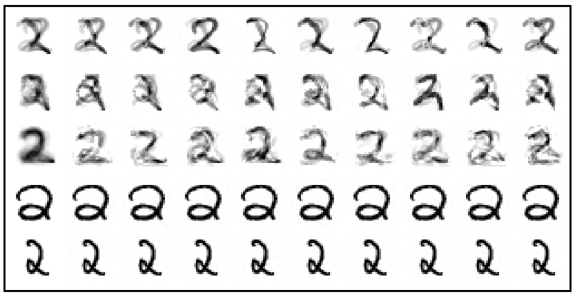

*Figure 3.25:* Direct modelling of the digits for $K = 30$ mixture components (mean vectors in 1st column).

In both illustrations each row represents generated digits from a single mixture component, where the 1st or leftmost digit is the component mean. For both $K = 10$ and $K = 30$ these mean digits are in most cases of reasonable quality with a faint background shadow. However both illustrations show poorly generated digits which hardly represents anything handwritten. In two cases the model has overfitted to a single digit with a very small variance. This is evident in row 5 (left) and 4 & 5 (right), where the same digit has been generated in all cases. This suggest the model complexity is to high due to overfit, but since we also model invalid dataspace decreasing the model complexity by using diagonal or isoptropical covariance matrix, we risk generating even more invalid digits. Hence choosing full covariance matrix seems like the right choice.

If we increase the amount of mixture components to $K = \{100, 200\}$ we get a set of generated digits shown in figures 3.26 - 3.27. From these digits the same conclusions can be made, where the generated digits are of poor quality with a few overfitted cases.

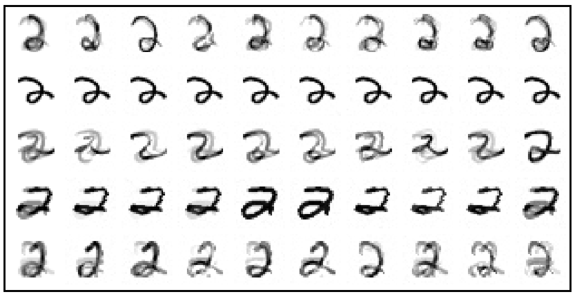

*Figure 3.26:* Direct modelling of the digits for $K = 100$ mixture components (mean vectors in 1st column).

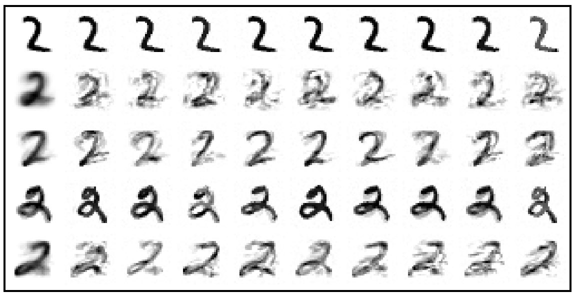

*Figure 3.27:* Direct modelling of the digits for $K = 200$ mixture components (mean vectors in 1st column).

Summary

The MoG model were used to model the handwritten digits directly without any feature extraction as pre-processing. With different amounts of mixture components the simulation clearly showed very poor ability to generate valid digits. This type of model is thus not very useful as a cognitive generative model.

## 3.5 Mixture of Gaussians Model

A different approach is to base the generative linear model given in (1.1) on modelling the sources $\mathbf{s}$ by a probability distribution $p(\mathbf{s})$. From $p(\mathbf{s})$ new random sources $\mathbf{s}$ can be extracted to generate a new digit $\mathbf{x}$ by using the linear model (1.1). To model the sources we employ the mixture of Gaussians model given by

$$p(\mathbf{s}) = \sum_{k=1}^{K} \alpha_k \cdot p(\mathbf{s}|\mathbf{\Theta}_k, k) \tag{3.3}$$

where each component $p(\mathbf{s}|\mathbf{\Theta}_k, k)$ is a multivariate Gaussian distribution defined in (2.30) and $\alpha_k$ denotes the component prior as before. To determine the mixture model parameters $\mathbf{\Theta}_k$ we again use the EM-algorithm as defined in section 2.4.

Model Complexity :

As discussed earlier the complexity of the model can be hard to optimize as it is a tradeoff between generating invalid or noisy digits as we have seen and achieving an overfitted model generating virtually the same digit multiple times. For the MoG model there are 3 levels of complexity of the covariance matrix, *isotropical*, *diagonal* and *full*, as shown in section 2.4.1. Eventhough this is a very coarse division, we conduct a few illustrative simulation to best find the optimal choice of complexity, shown in the following figures.

To better visualize the effect of the constraints figure 3.28 shows a scatterplot of the datapoints $\mathbf{s}_n$ projected onto the 1st and 2nd principal components (refer to section A.1). The estimated mixture components are shown as circles from to the isotropical covariance (refer to appendix A.4), where their relatively small radius is due to the restriction of neglecting covariance information. In addition it can be seen how the data is

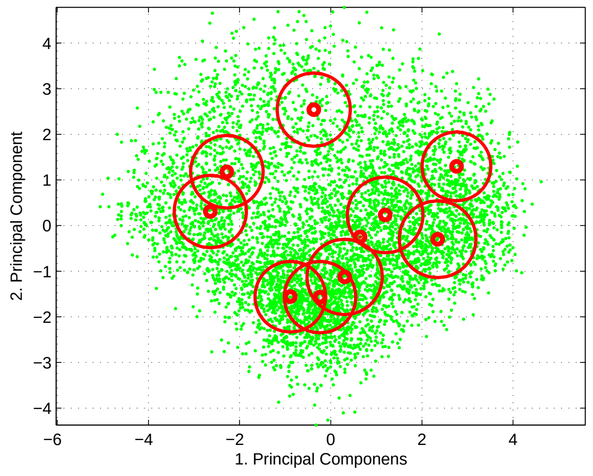

*Figure 3.28:* Convergence of mixture components with *isotropical* variance.

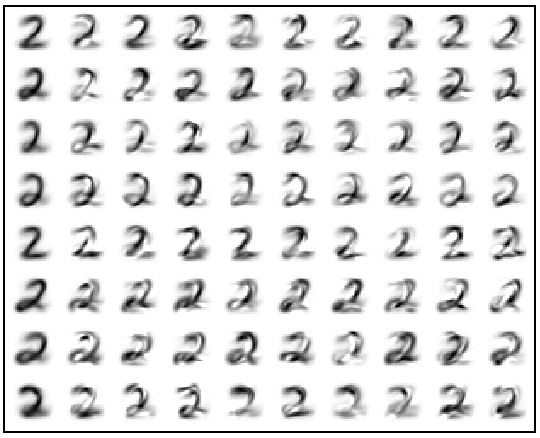

*Figure 3.29:* Generated digits from a MoG model with $K = 10$ components and *isotropical* covariance matrix (mean vectors in 1st column).

indeed uncorrelated, as information of one variable gives no information on the other. As the data and mixture components are projected into a 2D surface for visualization, information of modelling invalid data cannot be derived from this figure.

The right figure shows the corresponding generated digits, where the 1st column is the mean vector, $\mu_k$. Initially they seem very noisy and has many overlapping segments, but to compare the quality the results from the diagonal and full covariance matrix are shown in the following figures.

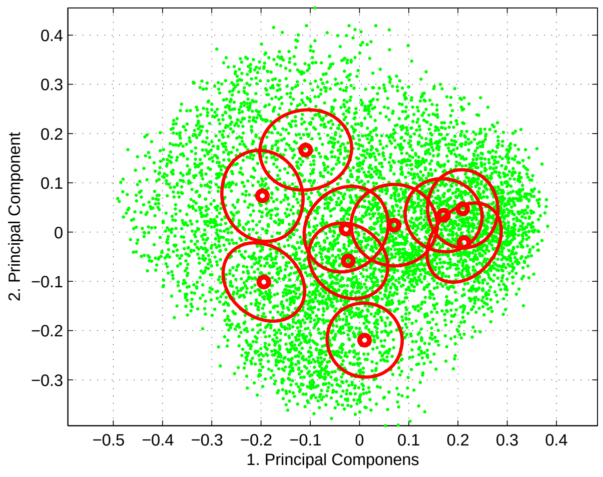

*Figure 3.30:* Convergence of mixture components with *diagonal* variance.

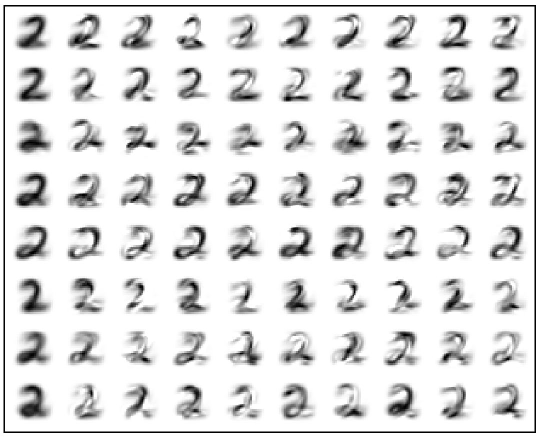

*Figure 3.31:* Generated digits from a MoG model with $K = 10$ components and *diagonal* covariance matrix (mean vectors in 1st column).

Figure 3.30 and 3.32 are the same type scatterplot, where the flexibility of the model becomes more evident. Especially for the full covariance matrix, which has a wide capture of the data structure. For the diagonal covariance matrix the elliptic shape is clear to see from the figure, but is not necessarily aligned with the axis due to the projection onto the 1st and 2nd principal component. In comparing all 3 scatterplots it is clear to see how the isotropical and diagonal covariances have relatively small radius. For the full covariance restriction, $\mathbf{\Sigma}^{(full)}$ is allowed to capture specific covariance directions, which leads to more elliptic shapes with higher radius.

The 3 right figures (3.29, 3.31 & 3.33) illustrate the effect of restricting the flexibility of the covariance matrix $\mathbf{\Sigma}$, where the 1st digit in each row is the corresponding mean vector, $\mu_k$. In comparing the generated digits from the three figures, the model with the isotropical covariance matrix (figure 3.29) shows noisy digits of poor quality with many overlapping segments. For the diagonal covariance matrix the quality has slightly improved, but is still to poor. The model with full covariance matrix (figure 3.33) also suffers from overlapping segments, but in decreasing numbers. For these particular generated digits no overfitted mixture components can be

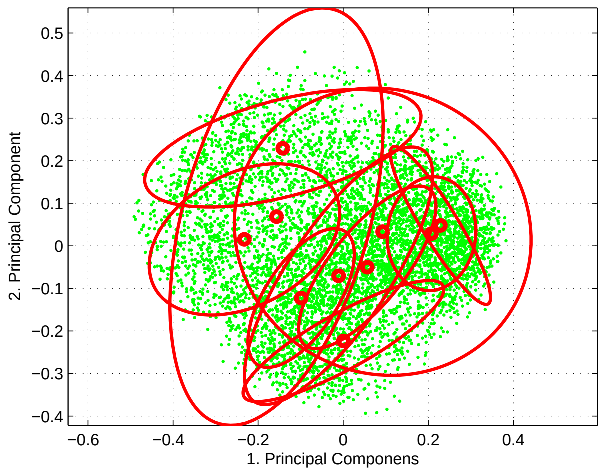

*Figure 3.32:* Convergence of mixture components with *full* variance.

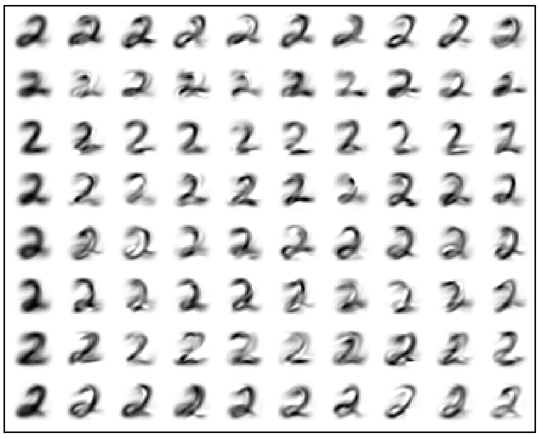

*Figure 3.33:* Generated digits from a MoG model with $K = 10$ components and *full* covariance matrix (mean vectors in 1st column).

identified, but this is mostly due to the relatively low amount of components $K = 10$, restricting the model from adapting to specific datapoints. Hence using no restrictions on the covariance matrix (i.e. full) for the MoG model is the preferred approach in our further simulations, which is presented next.

Results :

In the initial simulation a small codebook with $d = 30$ features is used for the 2 digit shown in figure 3.7. We conduct a set of simulations for different amount of mixture components $K = \{10, 20, 30, 50, 100, 200, 300, 400, 500\}$. Only a subset relevant for the analysis is presented here (data from all simulations can be found on the attached DVD). For the simulations we used a log-likelihood tolerance of $\Delta\mathcal{L} = 1e - 5$ as a stopping criteria and a maximum of $IT_{max} = 500$ iterations.

To evaluate the training of the MoG models for different sizes, figure 3.34 and 3.35 show the evolution of the log-likelihood $\mathcal{L}(\Theta) = \sum_n \ln p(\mathbf{x}_n|\Theta)$ as defined in (2.15) during the iterations of the "EM"-algorithm.

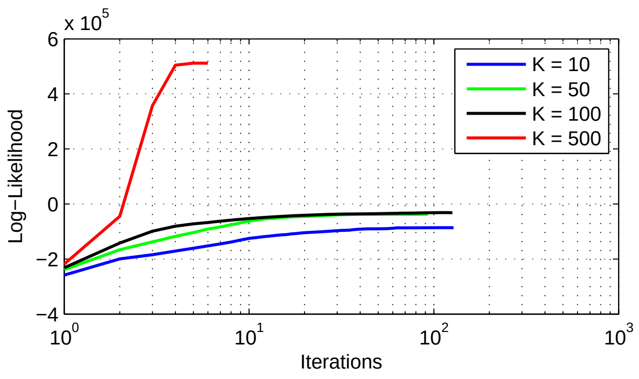

*Figure 3.34:* Evolution of the Log-Likelihood $\mathcal{L}(\Theta)$ for $K = \{10, 50, 100, 500\}$ and $d = 30$

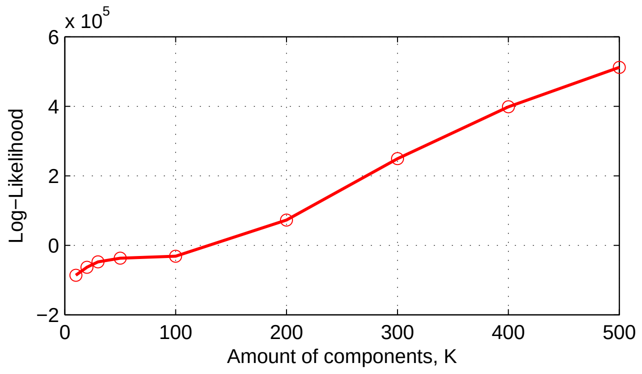

*Figure 3.35:* Final Log-likelihoods for different amount of components for $d = 30$.

The left figure depicts the evolution log-likelihood $\mathcal{L}(\Theta)$ during training of the model and how the "EM"-algorithm finally converges to a likelihood maximum. The different lengths is caused by the individual iterations needed to converge. The right figure show how the log-likelihood $\mathcal{L}(\Theta)$ increases with the amount of components as the model becomes more complex. Thus selecting a rather complex model is not necessarily desired as we might achieve an overfit to the training data and thus not model the underlying structure.

Figures 3.36 - 3.39 show a set of generated digits for only a subset of the component sizes $K$, where the 1st column in each figure is the corresponding component mean.

The top left figure with $K = 10$ mixture components reveals noisy digits as we have seen before similarly in figure 3.33. Increasing the amount of components to $K = 50, 100$ and $500$ to allow greater flexibility of the

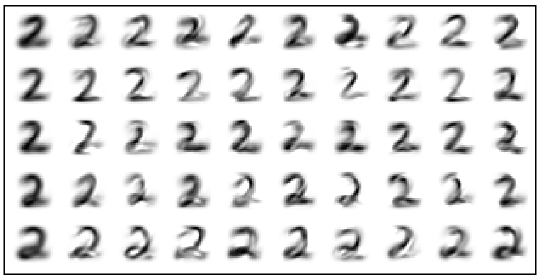

*Figure 3.36:* Generated digits from a MoG model with $d = 30$ features, $K = 10$ components and full covariance matrix (mean vectors in 1st column).

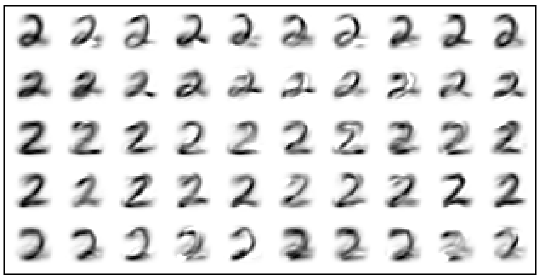

*Figure 3.37:* Generated digits from a MoG model with $d = 30$ features, $K = 50$ components and full covariance matrix (mean vectors in 1st column).

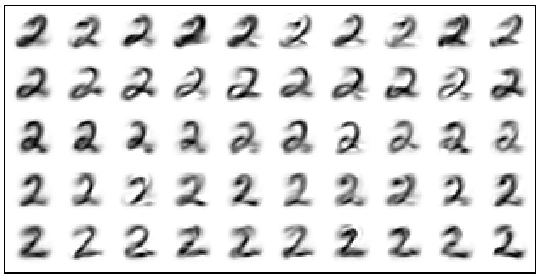

*Figure 3.38:* Generated digits from a MoG model with $d = 30$ features, $K = 100$ components and full covariance matrix (mean vectors in 1st column).

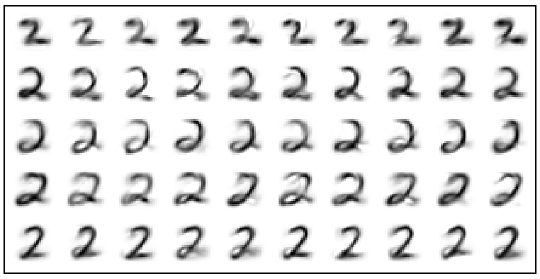

*Figure 3.39:* Generated digits from a MoG model with $d = 30$ features, $K = 500$ components and full covariance matrix (mean vectors in 1st column).

MoG model do generate less noisy digits with fewer overlapping segments. With $K = 500$ mixture components the MoG model tends to overfit, but we still generate sufficiently different digits. Eventhough these digits are of fairly good quality, the large amount of components yields a poorly estimated covariance matrix as discussed previously due to the curse of dimensionality $(5958/500 = 11.9$ samples/dim.) and simply fails to capture the essential correlation between the codebook elements as also discussed earlier.

Instead we increase the size of the codebook $\mathbf{A}$ to $d = 100$ and re-simulate with the same stopping criteria. Again the amount of components is set to $K = \{10, 20, 30, 50, 100, 200, 300, 400, 500\}$ and only present a subset of these. Initially the evolution of the "EM"-algorithm in training the MoG models is shown in figures 3.40 and 3.41.

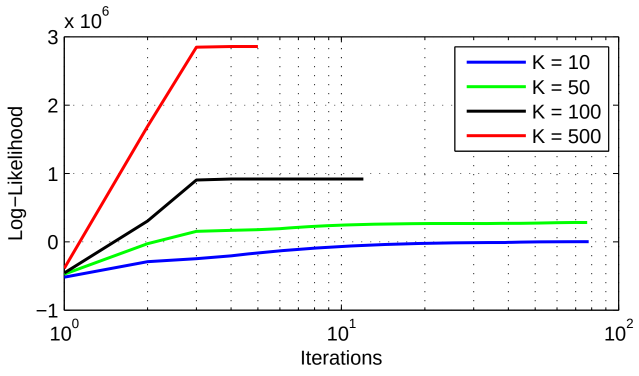

*Figure 3.40:* Evolution of the Log-Likelihood $\mathcal{L}(\boldsymbol{\Theta})$ for $K = \{10, 50, 100, 500\}$ and $d = 100$ features.

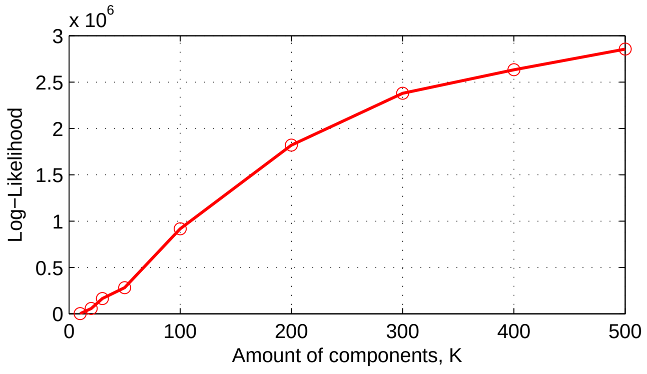

*Figure 3.41:* Final Log-likelihoods for different amount of components for $d = 100$ features.

The left figure reveals a successfully converged "EM"-model as before. The right figure depicts how the log-likelihood $\mathcal{L}(\boldsymbol{\Theta})$ increases as the MoG model becomes more complex as we have seen before. The corresponding generated digits are shown in figures 3.42 - 3.45, where the 1st column is the mean vector as before.

The top left figure for $K = 10$ reveals noisy digits with many overlapping segments similarly as for $d = 30$.

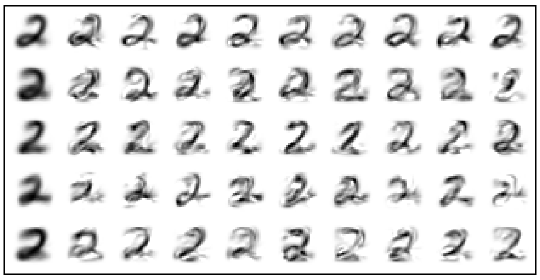

*Figure 3.42:* Generated digits from a MoG model with $d = 100$ features, $K = 10$ components and full covariance matrix (mean vectors in 1st column).

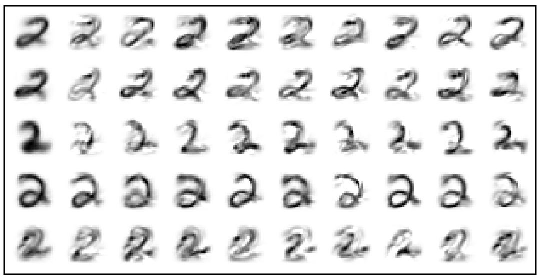

*Figure 3.43:* Generated digits from a MoG model with $d = 100$ features, $K = 50$ components and full covariance matrix (mean vectors in 1st column).

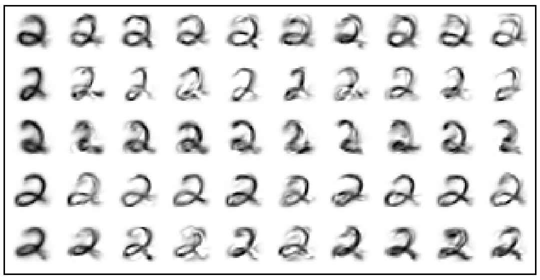

*Figure 3.44:* Generated digits from a MoG model with $d = 100$ features, $K = 100$ components and full covariance matrix (mean vectors in 1st column).

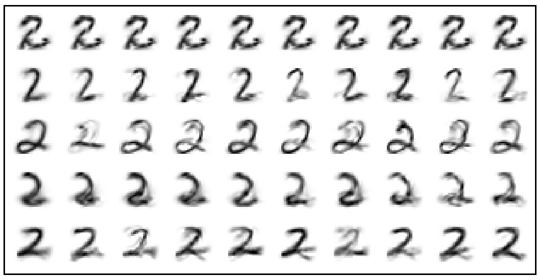

*Figure 3.45:* Generated digits from a MoG model with $d = 100$ features, $K = 500$ components and full covariance matrix (mean vectors in 1st column).

Increasing $K$ results in digits with less noise, but not with equal quality as for $d = 30$ in previous figures 3.36 - 3.39. With the larger codebook $d = 100$, the model parameters $\mathbf{\Theta}$ become harder to estimate with sufficient precision, especially the covariance matrix as discussed earlier. This is evident to see for $K = 500$ in figure 3.45, where model clearly suffers from overfit with little variance. The few digits, which are not similar to the mean vector (leftmost digit in each row) are of very poor quality, due to the poorly estimated covariance matrix.

Multilayered MoG model :

Based on the MoG model in (3.3) we can also form dual-layered MoG model by modelling each mixture component with $k'$ sub mixtures. The dual-layer MoG model can be expressed as

$$p(\mathbf{s}) = \sum_{k=1}^{K} \alpha_k \cdot p(\mathbf{s}|\mathbf{\Theta}_k, k) = \sum_{k=1}^{K} \alpha_k \cdot \left[ \sum_{k'=1}^{K'} \alpha_{k'} \cdot p(\mathbf{s}|\mathbf{\Theta}_{k'}, k') \right] \tag{3.4}$$

As the datasample assignments for each mixture component are soft for the MoG model we must associate the datasamples by hard-assignment for the top-layer mixtures and use these samples as the dataset for each sublayer mixture component. Incorporating the second layer gives us a second prior $\alpha_{k'}$ to attenuate the mixture components with few samples associated (refer to eq. (2.46)). However if we neglect the prior for the sublayer $\alpha_{k'}$ then this model is exactly the same as a single-layer MoG model with $kk'$ components.

As just shown the modelling capabilities for the single-layer MoG model was not sufficient for cognitive data, we do not expect any multilayered MoG model to outperform the single-layered MoG model.

Summary :

Based on our linear generative model $\mathbf{x} = \mathbf{A}\mathbf{s}$, the sources $\mathbf{s}$ were modelled using a MoG density estimator for different amount of components $K$ and codebook sizes $d$. For the small codebook $d = 30$, this approach generated noisy digits for low $K$. As $K$ increased the quality of the digits improved, but due to the limited size of the dataset $N$ the model parameters for the high value of $K$ became poorly estimated. The model simply failed to capture the essential correlation in the sources $\mathbf{s}$. In essence it becomes a tradeoff between biased model yielding noisy digit and an overfitted model with poor parameters. Increasing the codebook to $d = 100$ only amplified the problem of estimating the model parameters with sufficient precision.

Thus this model does unfortunately not have the ability to generate digits of sufficient quality.

## 3.6 Mixture of Factor Analyzers

One of the major limitations on the MoG model is the rather coarse restrictions available as we have seen. Using the most flexible approach (full covariance matrix) resulted in poorly estimated model parameters and thus unsatisfactory generated digits. As an improvement we introduced the mixture of factor analyzers (MFA) earlier in section 2.5 as a mixture model with dimension reduction capabilities. In this section we employ the MFA algorithm to model the sources $\mathbf{s}_n$. The model can be written as

$$p(\mathbf{s}) = \sum_{k=1}^{K} \alpha_k \cdot p(\mathbf{s}|\mathbf{\Theta}_k, k) \tag{3.5}$$

where $p(\mathbf{s}|\mathbf{\Theta}_k, k) = \mathcal{N}\{\mu_k, \mathbf{\Lambda}_k\mathbf{\Lambda}_k^T + \mathbf{\Psi}\}$ is the factor analyzer as defined earlier with model parameters $\mathbf{\Theta}_k = \{\alpha_k, \mu_k, \mathbf{\Lambda}_k\mathbf{\Psi}\}$ and $\alpha_k$ denotes the component prior. The model is trained using the "EM"-algorithm as defined in section 2.5 (refer to the associated DVD for the `MATLAB` code) [8].

Compared to the MoG model the MFA model has the extra input parameter $d'$ defining the dimension reduction. This parameter is especially important as it may allow the model to capture cognitive correlations in sub-dimensional dataspace as opposed to the MoG model. Setting $d'$ too low may cause this cognitive correlation to be projected onto a sub-dimension too low preventing efficient modelling. In contrast if $d'$ approach the original dimension of the dataspace and hence may suffer from the same problem of modelling sufficiently as the MoG model did.

Results :

For the simulations we use two different cookbook sizes of cognitive features $d = \{30, 100\}$ and a wide range of amount of mixture components $K = \{10, 20, 30, 50, 100\}$. For the reduction factors we use $d' = \{5, 8, 10, 15, 20, 25\}$ to cover a wide range. Only a subset of results will be presented here relevant for the analysis, please refer to the DVD for a complete simulation results reference.

In the initial simulation we use a codebook of size $d = 30$, use $K = 10$ mixture components and evaluate for all different reduction factors $d' = \{5, 8, 15, 25\}$. From the trained model a set of digits were generated (refer to appendix A.5) and evaluated visually.

For the simulations we used a $\Delta\mathcal{L} = 1 \times 10^{-5}$ as a stopping criteria and a maximum of $500$ iterations allowed. This initialization of the MFA model is used for all further simulations unless otherwise noted. The `MATLAB` code for the simulations can be found on the DVD. Figures 3.46 and 3.47 show the training of the MFA model for different reduction factors $d'$.

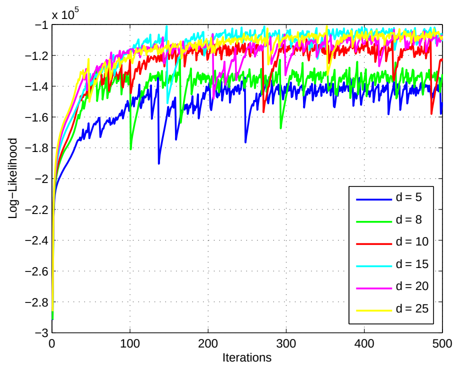

*Figure 3.46:* Evolution of the Log-Likelihood $\mathcal{L}(\mathbf{\Theta})$ for $d = 30$ and $d' = \{5, 8, 10, 15, 20, 25\}$

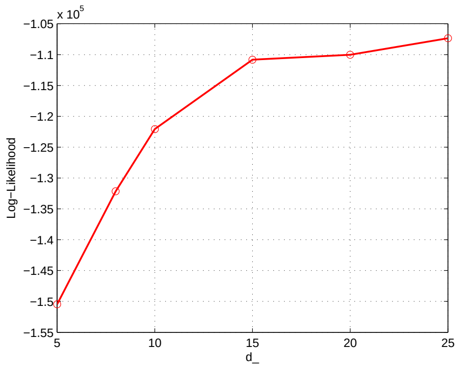

*Figure 3.47:* Log-likelihoods for different values of dimension reduction for $d = 30$.

[8] Available from the website of Prof. Zoubin Ghahramani at http://learning.eng.cam.ac.uk/zoubin/software.html

The left figure 3.46 shows how the log-likelihood has successfully converged for all $d'$. The noise on the flat asymptote is due to $\mathcal{LL}$ violations, where the log-likelihood failed to increase during iterations. In those cases the MFA algorithm re-iterated until the likelihood increased. The relatively high noise level has also caused the algorithm to fail to meet the stopping criteria until the maximum number of iterations allowed was met. This indicates an instability in the implementation of the MFA algorithm, but since the algorithm is considered converged, we have done no further investigations to remedy this issue. The right figure 3.47 reveals how the likelihood increase as the model becomes more complex for increasing $d'$, as expected.

Figures 3.48 - 3.51 show a set of generated digits for only a subset of the component sizes $K$, where the 1st column in each figure is the corresponding component mean vector, $\mu_k$.

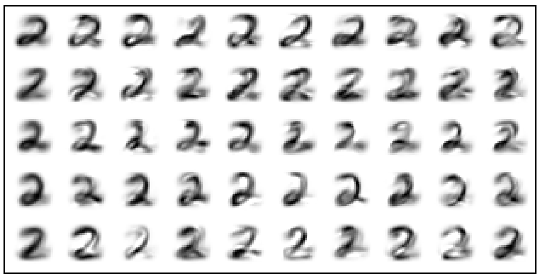

*Figure 3.48:* Generated digits from a MFA model with $d = 30$ features, $K = 10$ components and a reduction to $d' = 5$ dimensions (mean vectors in 1st column).

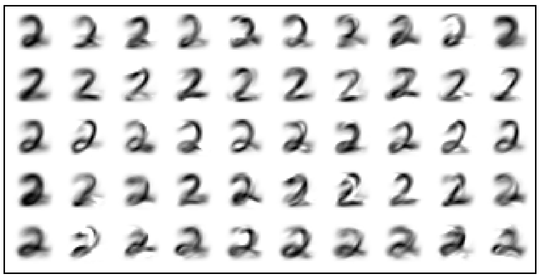

*Figure 3.49:* Generated digits from a MFA model with $d = 30$ features, $K = 10$ components and a reduction to $d' = 8$ dimensions (mean vectors in 1st column).

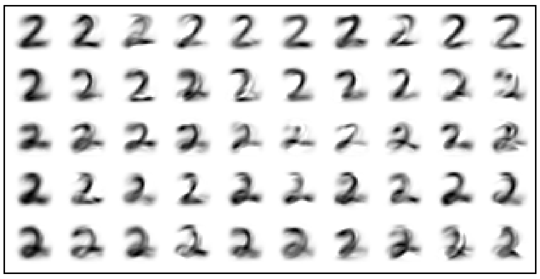

*Figure 3.50:* Generated digits from a MFA model with $d = 30$ features, $K = 10$ components and a reduction to $d' = 15$ dimensions (mean vectors in 1st column).

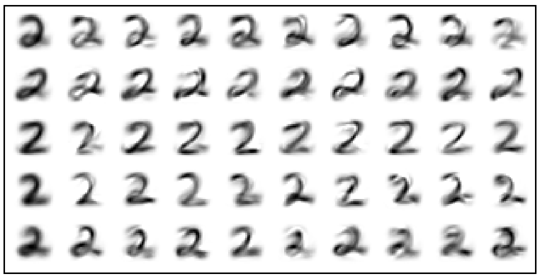

*Figure 3.51:* Generated digits from a MFA model with $d = 30$ features, $K = 10$ components and a reduction to $d' = 25$ dimensions (mean vectors in 1st column).

The generated digits in the top left figure 3.48 with the lowest dimension $d' = 5$ are of fairly good quality with a variance sufficiently large to generate different digits, still with a boldface style. Increasing to $d' = 8$ gives slightly more noisy digits as we have seen before for the MoG model and for $d' = 15$ the digits become worse, as can be seen in figure 3.50. For $d' = 25$ the dimension of the modelled data is close to the original, which is evident from figure 3.51, where the we have failed to capture the underlying cognitive structure of the data, similar to the MoG model.

If we increase the amount of mixture component $K = 30$, the evolution of the log-likelihood is shown in figure 3.52 and 3.53. These figures show the same evolution as for $K = 10$ as expected. From figure 3.52 the noise on the flat asymptote is present again, but since the algorithm has converged successfully we do not consider this a problem for the simulation. The corresponding generated digits are illustrates in figures 3.54 - 3.57.

The generated digits shown all suffer from the noise of overlapping segments and boldface type as we have seen before and for increasing amount of clusters $K = \{50, 100\}$ any generated digit suffer from these defects in increasing numbers (these results are not shown, but can be found on the DVD). In fact simulations for the larger codebook with $d = 100$ cognitive features, the same defects occur to an even greater extent due to the increased number of parameters to determine for the model.

Summary :

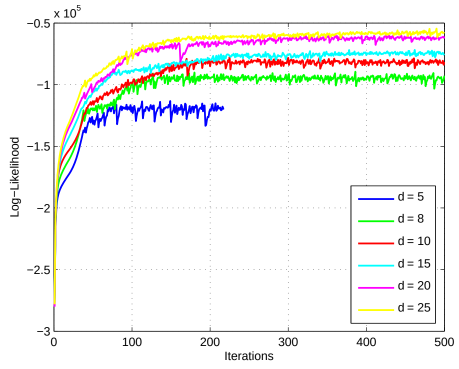

*Figure 3.52:* Evolution of the Log-Likelihood $\mathcal{L}(\Theta)$ for $d' = \{5, 8, 10, 15, 20, 25\}$

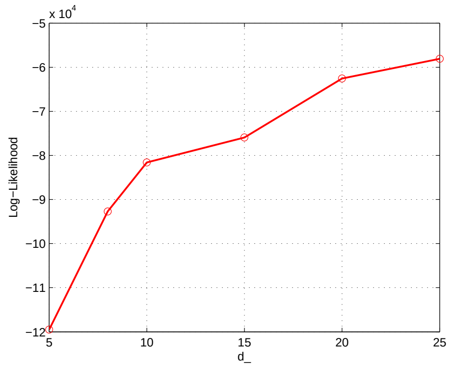

*Figure 3.53:* Log-likelihoods for different values of dimension reduction.

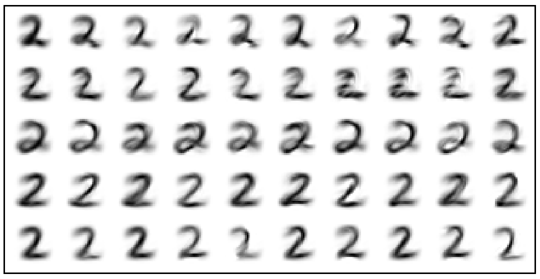

*Figure 3.54:* Generated digits from a MFA model with $d = 30$ features, $K = 30$ components and a reduction to $d' = 5$ dimensions (mean vectors in 1st column).

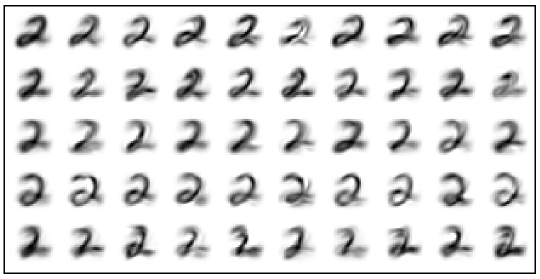

*Figure 3.55:* Generated digits from a MFA model with $d = 30$ features, $K = 30$ components and a reduction to $d' = 8$ dimensions (mean vectors in 1st column).

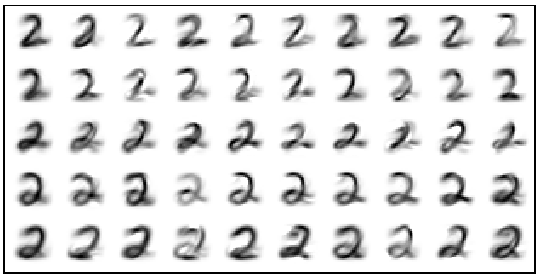

*Figure 3.56:* Generated digits from a MFA model with $d = 30$ features, $K = 30$ components and a reduction to $d' = 15$ dimensions (mean vectors in 1st column).

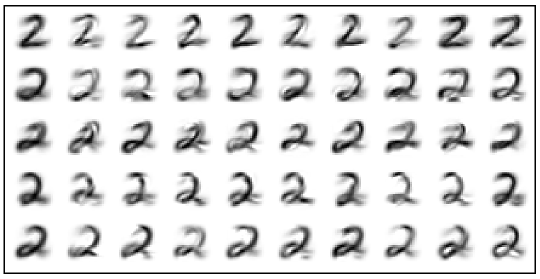

*Figure 3.57:* Generated digits from a MFA model with $d = 30$ features, $K = 30$ components and a reduction to $d' = 25$ dimensions (mean vectors in 1st column).

Introducing the MFA model with dimension reduction capabilities was done to allow an improved capture of any cognitive correlation in the image dataset and to avoid the poor estimated covariance information observed for the MoG model. In the simulations digits with improved quality compared to the MoG model was generated, but were still suffering from the defects of noise and overlapping segments. In general this type of MFA model cannot be characterized as a sufficient cognitive model.

## 3.7 Summary

In section 2 the class of mixture models was introduced and described as a basis of generative linear models, $\mathbf{x} = \mathbf{A}\mathbf{s}$. In this part we have formed a set of generative models based on clustering and evaluated their performance in terms of digit generation for the MNIST dataset. First a model using K-Means to cluster codebook features to assemble representative digits revealed very poor performance in generating handwritten digits as the model failed to capture any underlying correlation between the features.

Secondly the mixture of Gaussians model were introduced and showed improved generated digits, but suffered from poorly estimated covariance matrices depending on model complexity leading to poorly generated handwritten digits in either case.

As a third model type, the mixture of factor analyzers were presented as an advancement to the MoG model to better capture the sub-dimensional correlation between cognitive features. On the MNIST dataset this model type did make a small improvement on the quality of generated digits, but suffered from the same defects as the MoG model.

In general the class of mixture models with radial kernel functions are thus not sufficiently advanced or complex to capture the correlation of cognitive features for handwritten digits as data.

# 4. Deep Network Models

A different approach for probability density estimation is based on non-linear multilayered networks or *Deep Networks* as introduced earlier used for both generation and as discriminative functions. In this section we introduce a multilayered network model based on a non-linear matrix factorizations described in the following section. Before we describe our own deep network model we introduce the concept of deep belief nets based on Hinton et al. [12]

## 4.1 Hinton's Deep Belief Nets

A deep belief network is formed by $L$ individual network layers each with a set of visible units $\mathbf{v} = \{v_1, v_2, \ldots\}$ and hidden units $\mathbf{h} = \{h_1, h_2, \ldots\}$. The belief net can be probabilistically modelled for an observed vector $\mathbf{x}$ by the joint distribution

$$p(\mathbf{x}, \mathbf{h}^1, \mathbf{h}^2, \ldots, \mathbf{h}^L) = p(\mathbf{x}|\mathbf{h}^1)p(\mathbf{h}^1|\mathbf{h}^2)\cdots p(\mathbf{h}^{L-2}|\mathbf{h}^{L-1})p(\mathbf{h}^{L-1}, \mathbf{h}^L) \tag{4.1}$$

where $\mathbf{h}^l$ represent the hidden units at layer $l$.

### 4.1.1 Restricted Boltzmann Machine

In the logistic belief net (DBN) used by Hinton et al. the net is composed by stochastic binary units and the individual layers are modelled by a *Restricted Boltzmann Machine* (RBM) expressed by the likelihood

$$p(h_j^l = 1|\mathbf{h}^{l+1}) = \mathrm{sigm}\Big( - b_j^l - \sum_{k=1}^{n^{l+1}} A_{kj}^l h_k^{l+1}\Big) \tag{4.2}$$

where $\mathrm{sigm}(x) = 1/(1+\exp(-x))$, $b_j^l$ is the bias for unit $j$ in layer $l$ and $\mathbf{A}^l$ denote the weight matrix for layer $l$. The top-level prior $p(\mathbf{h}^{L-1}, \mathbf{h}^L)$ has a joint distribution given by

$$p(\mathbf{h}^{L-1}, \mathbf{h}^L) = \frac{1}{Z} \exp\Big( - (\mathbf{h}^L)^T \mathbf{A}(\mathbf{h}^{L-1}) - \mathbf{b}^T(\mathbf{h}^{L-1}) - \mathbf{c}^T(\mathbf{h}^L)\Big) \tag{4.3}$$

where $\mathbf{A}$ is the weight matrix and $\mathbf{b}$ and $\mathbf{c}$ are the biases for visible and hidden units respectively.

### 4.1.2 Complementary Priors

For a generative DBN the likelihood of observed data $p(\mathbf{x}|\mathbf{h})$ can easily be computed by (4.2). One of the difficult aspects of inferring the posterior of the hidden variables $p(\mathbf{h}|\mathbf{x})$ in directed belief nets is dependency between the individual hidden units $\{h_1, h_2, \ldots\}$. This dependency is also known as the phenomenon of 'explaining away', where if one event has explained an observation, then it 'explains' away all other possible explainable events for that particular observation [12].

One of the most important properties of Hinton's DBN model is the introduction of *complementary priors* to eliminate this 'explaining away', where such priors ensures the posterior $p(\mathbf{h}^{l+1}|\mathbf{h}^l)$ factorizes and thus induces independence, i.e. $p(\mathbf{h}^{l+1}|\mathbf{h}^l) = \prod_j p(\mathbf{h}_j^{l+1}|\mathbf{h}^l)$. It is not explicitly clear from Hinton et al. how these complementary are implemented (refer to [12]), except the practical induction of the complementary priors for DBN give rise to a posterior calculated as

$$p(h_j = 1|\mathbf{v}) = \mathrm{sigm}\Big( - c_j - \sum_k A_{jk} v_j\Big) \tag{4.4}$$

Hence the transposed weight matrix $\mathbf{A}^T$ is used to infer the posterior of the hidden variables. This is a very powerful results as it allows us to sample easily from both the likelihood $p(\mathbf{v}|\mathbf{h})$ and posterior $p(\mathbf{h}|\mathbf{v})$ using these tied weights.

### 4.1.3 Infinite belief network

For an infinite directed model with tied weight we achieve complementary priors at all layers as shown in figure 4.1. Due to the tied weights this model is an equivalent to a single layer RBM described above.

To generate data from an RBM corresponds to extracting samples from $p(\mathbf{v})$, where we start with a random state in one of the layers, visible or hidden. By performing alternating Gibbs sampling, i.e. iterating between extracting from $p(\mathbf{v}|\mathbf{h})$ and $p(\mathbf{h}|\mathbf{v})$ (refer to appendix A.7), and continue until we sample from the equilibrium distribution, we conduct the same operation as generating data from the infinite belief net with tied weights. Each full step in the Gibbs sampling process is thus equivalent to computing the exact posterior distribution in a layer of the infinite logistic belief net.

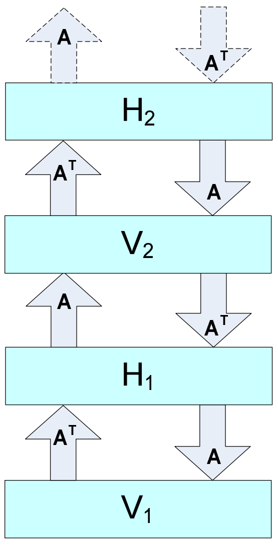

*Figure 4.1:* Infinite directed logistic net with tied weights at each layer equivalent to single layer RBM.

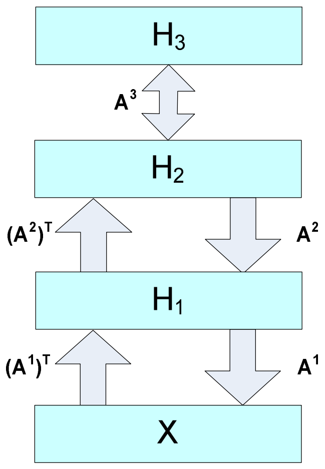

*Figure 4.2:* The deep belief network with undirected connections between top two layers, where the generative and recognition weights are tied for each layer.

### 4.1.4 Learning

The weights of an RBM can be learned unsupervised from maximizing the likelihood function and for a single weight $a_{ij}^{11}$ from unit $j$ in layer $H_1$ to unit $i$ in layer $V_1$ the gradient of the log can be expressed as [12]

$$\frac{\partial \log p(\mathbf{v}^1)}{\partial a_{ij}^{11}} = \mathcal{E}\{h_j^1(v_i^1 - v_i^2)\} \tag{4.5}$$

This shows how the update of a weight is only dependent on local data, i.e. the input and output associated to that weight. Due to the equivalence to the infinite belief net this derivative can be collapsed for all weights and layers to the difference between the average correlation of the top and bottom layers of the infinite net [12]

$$\frac{\partial \log p(\mathbf{v}^1)}{\partial a_{ij}} = \mathcal{E}\{v_i^1 h_j^1\} - \mathcal{E}\{v_i^\infty h_j^\infty\} \tag{4.6}$$

where the term $\mathcal{E}\{v_i^\infty h_j^\infty\}$ corresponds to the top layer of the infinite belief net and can be found by running Gibbs sampling on the RBM until it reached equilibrium distribution. In addition (4.6) also shows that if the RBM models the true distribution of the visible data $\mathbf{v}^1$, the gradient becomes zero.

Greedy Learning

In Hinton's approach the *contrastive divergence* is used instead of ML learning, where the top layer correlation is measured after only $n$ steps of the Markov chain Gibbs sampling. This corresponds to ignoring the derivatives of all layers above $n$ and thereby induces quantization. It can further be shown that the contrastive divergence learning it equivalent to minimizing the Kullback-Leibler divergence $\mathrm{KL}\big(p^1 || p_\theta^\infty\big) - \mathrm{KL}\big(p_\theta^n || p_\theta^\infty\big)$ [12].

Applying contrastive divergence in the obvious way to a belief net with different weights at each layer the algorithm takes far too long to reach the conditional equilibrium state with a clamped data vector and thus renders practical useless. Instead Hinton et al. suggest a two step procedure for training a deep belief network with untied weights, initially with a greedy algorithm to achieve approximate weights for a subsequent step of fine-tuning all weights.

The idea behind the greedy algorithm is to allow each layer in the sequence to receive a different representation of the data and figure 4.2 illustrates a hybrid belief net, where the top two layers are undirected connected. This corresponding to modelling $p(\mathbf{h}^4, \mathbf{h}^5, \ldots, \mathbf{h}^\infty)$ in this case and is thus equivalent to have infinitely many higher layers with tied weights.

For inferring $\mathbf{A}^1$ the greedy algorithm assumes the weights for higher layers will be used to construct a complementary prior for $\mathbf{A}^1$. This is equivalent to assuming that all the weight matrices are constrained to be equal. Once $\mathbf{A}^1$ has been learned, the data can be mapped through $(\mathbf{A}^1)^T$ to create higher-level 'data' at the first hidden layer. The unsupervised greedy algorithm can be summarized in table 4.1.

```
1. Learn $\mathbf{A}^1$ assuming all weight matrices are tied.

2. Freeze $\mathbf{A}^1$ and use $(\mathbf{A}^1)^T$ to infer the factorial approximate posterior distributions over hidden variables in the first layer, even if subsequent changes in higher level weights mean that this inference method is no longer correct.

3. Keeping all the higher weight matrices tied to each other, but untied from $\mathbf{A}^1$, learn an RBM model of the higher-level 'data' that was produced by using $(\mathbf{A}^1)^T$ to transform the original data.
```

*Table 4.1:* Pseudocode for greedy learning algorithm

For equal sized layers subsequent layers are initialized with the weights learned from the parent layer. It can be shown that the greedy algorithm induces a variational bound on the negative log probability of a data vector $\mathbf{v}^1$ ensuring adding a new layer will never decrease the log probability of the data under the full generative model [12].

Fine Tuning

After completing the greedy learning of the weights in all layers, neither the higher layer weights nor the simple inference procedure are optimal for the lower layers. The fine-tuning algorithm proposed as the second learning step is conducted to revise the weights that were learned first to better fit with the weights that were learned later (top layer).

For the fine-tuning all weights are untied within layers, but retain the restriction that the posterior in each layer must be approximated by a factorial distribution in which the variables within a layer are conditionally independent given the values of the variables in the layer below. Hinton et al. uses a variant of the wake-sleep algorithm, where the generative weights are updated during a 'wake' phase and the recognition weights are updates during the 'sleep' phase.

In the up-pass step the recognition weights are used to stochastically choose a state for every hidden variable. The generative weights on the directed connections are then adjusted using the maximum likelihood learning rule in eq. (4.5). The weights on the undirected connections at the top level are learned as before by fitting the top-level RBM to the posterior distribution of the previous layer. The 'down-pass' initiates with a state of the top-layer and uses the generative weights to stochastically activate each lower layer in turn. By initializing the top layer with an up-pass step and then only allowing a few iterations of alternating Gibbs sampling before initiating the down-pass, a 'contrastive' form of the wake-sleep algorithm is achieved, which eliminates the need to sample from the equilibrium distribution of the associative memory. For a more detailed description of the fine tuning algorithm refer to Hinton et al. [12].

### 4.1.5 Implementation

In the practical formation of the deep belief net Hinton et al. augments a classifier network at the top-level associative memory and thus uses supervised training to learn data labels. The labels is represented by turning on one unit in a 'softmax' group of 10 units, i.e. $\mathbf{c} = \{0 \dots 1 \dots 0\}$. Classifying an observed data is conducted by propagating the data up through the layers and here Hinton et al. proposes different methods for evaluating the class relation. Refer to [12] for details.

To generate class conditional samples from the model the desired label $\mathbf{c} = \{0 \dots 1 \dots 0\}$ is clamped and a sample is then produced by conducting alternating Gibbs sampling in the top-level associative memory $p(\mathbf{v}, \mathbf{h}|\mathbf{c})$ until the equilibrium distribution is reached. The sample is then propagated down through the layers in a single downpass.

This short outline of the deep belief network model introduced by Hinton et al. [12] serves as motivation for our simpler deep network model base on only unsupervised greedy learning. Throughout the detailed description of our model we will note the essential differences for better understanding on the model simplifications.

## 4.2 The Network Model

Motivated by the deep belief net by Hinton et al. we propose a simpler hierarchical generative model build from individual modules as opposed to using the same model for each layer as Hinton et al. does.

A deep network model is built on hierarchical layered non-linear functions. If we consider $N$ observed $M$ dimensional datavectors $\mathbf{X} = \{\mathbf{x}_n\}_{n=1}^{N}$, we base the generative deep net model on the non-linear generalized factorization of a single observation given by

$$\mathbf{x} = \mathbf{A}f(\mathbf{s}) \tag{4.7}$$

where the columns of $\mathbf{A} = \{\mathbf{a}_r\}_{r=1}^{d}$ are the $M$ dimensional feature vectors and $f(\mathbf{s})$ is the non-linear mapping function of the $d$ dimensional sources $\mathbf{s}$. This gives a far more adaptable model, where the linear factorization in (1.1) now becomes a special case with $f(\mathbf{s}) = \mathbf{s}$. In such case an $L$ multilayered architecture could be collapsed into an equivalent 1-layer structure due to the linearity, i.e. $\mathbf{x} = \mathbf{A}^1 f_1(\mathbf{A}^2 f_2(\dots\mathbf{A}^L f_L(\mathbf{s})))$ and thus the multilayered structure is no longer appropriate. Introducing the non-linearity is hence essential for deep network models.

To induce greater flexibility of our non-linear model (4.7) we add an extra bias or threshold term. On vector form this can be written as $\mathbf{x} \approx \mathbf{A}f(\mathbf{s}) + \mathbf{b}$, where $\mathbf{b}$ is an $M$-dimensional vector. By augmenting $\mathbf{A}$ and the vectors $\mathbf{s}$ the threshold $\mathbf{b}$ can be included in matrix $\mathbf{A}$ by

$$\mathbf{A} \leftarrow \left[\, \mathbf{A}\ \mathbf{b} \,\right] \qquad \wedge \qquad \mathbf{s}_n \leftarrow \begin{bmatrix} \mathbf{s}_n \\ 1 \end{bmatrix} \tag{4.8}$$

i.e. $\mathbf{A}$ is now a $M \times d+1$ matrix and $\mathbf{s}$ a $d+1$ dimensional vector. For a single observed sample $\mathbf{x}$ this generative model can graphically be illustrated in the network structure as depicted in figure 4.3.

From our non-linear model (4.7) we can model the observed data $\mathbf{x}$ probabilistically by expressing $p(\mathbf{x})$ as a marginalization over the sources $\mathbf{s}$, written as

$$p(\mathbf{x}) = \int p(\mathbf{x}, \mathbf{s})\ \mathrm{d}\mathbf{s} = \int p(\mathbf{x}|\mathbf{s})p(\mathbf{s})\ \mathrm{d}\mathbf{s} \tag{4.9}$$

where $p(\mathbf{s})$ is the unknown distribution of the sources, which for our deep network is modelled in subsequent layers by the likelihood $p(\mathbf{s}^l|\mathbf{s}^{l+1})$. Thus for an $L$ layered model $p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{s}^1)p(\mathbf{s}^1|\mathbf{s}^2)\dots p(\mathbf{s}^L)\ \mathrm{d}\mathbf{s}$.

This is a somewhat different model than Hinton's belief net in (4.1), where we marginalize over the sources $\mathbf{s}$ at the top-layer instead of using an undirected associative memory modelling a joint distribution. This means

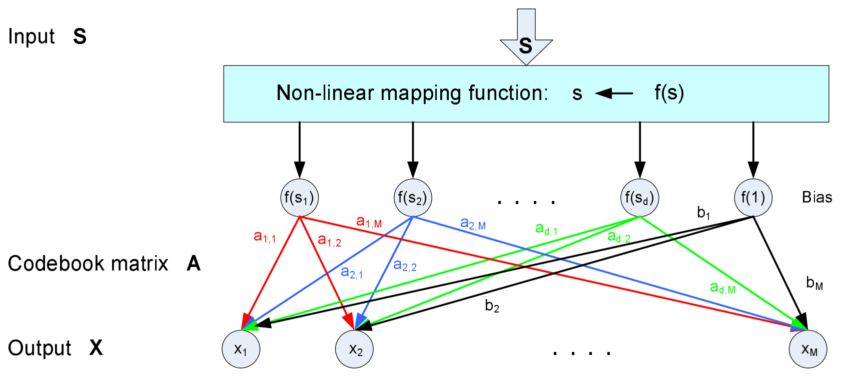

*Figure 4.3:* Network structure of modelling a single datasample. The colored weights illustrate the column vectors of the codebook matrix, $\mathbf{a}_r$

we do not implement complementary priors and thus cannot compute the posterior $p(\mathrm{s}^{l+1}|\mathrm{s}^l)$ by performing an up-pass step with the transposed weight matrix $\mathbf{A}^T$ as easy as Hinton, as we shall see.

From the linear decomposition wrt. $f(\mathrm{s})$ in (4.7) we have implicitly assumed additive Gaussian noise (refer to appendix A.6 for details) as we did for NMF in section 2.1. We can therefore express the likelihood of the input $\mathrm{s}_{l-1}$ for any layer as the probability conditioned on the sources given by the Gaussian distribution

$$p(\mathrm{s}^{l-1}|\mathrm{s}^l) = \frac{1}{\sqrt{(2\pi)^M|\sigma^2\mathcal{I}|}} \exp\Big(-\frac{1}{2\sigma^2}(\mathrm{s}^{l-1} - \mathbf{A}f(\mathrm{s}^l))^T(\mathrm{s}^{l-1} - \mathbf{A}f(\mathrm{s}^l))\Big) \tag{4.10}$$

where the covariance $\sigma^2$ is an $M$ dimensional vector, i.e. the covariance matrix is assumed diagonal with all correlation in $\mathbf{x}$ represented in the factorization $\mathbf{A}\mathrm{s}$. This notation in (4.10) means the initial layer with input $\mathbf{x}$ is a special case, where $\mathbf{x} = \mathrm{s}_{l-1}$.

For the top-level prior $p(\mathrm{s})$ we assume independent priors, i.e. $p(\mathrm{s}) = \prod_i p(s_i)$ and model the sources by the Laplace distribution given by

$$p(\mathbf{s}) = \frac{\alpha}{2}\exp(-\alpha|\mathbf{s}|) \tag{4.11}$$

The Laplace distribution is illustrated in figure 4.4.

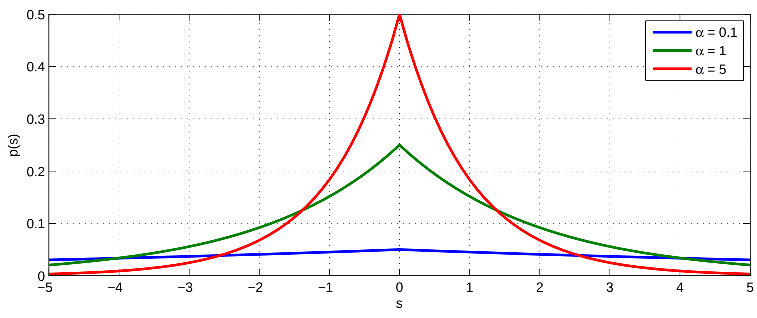

*Figure 4.4:* The Laplace distribution for different scale parameters.

From both the definition in (4.11) and figure 4.4 is it evident to see that for large scale parameters $\alpha$ only small value of s are encouraged. In contrast for $\alpha = 0$ we achieve the uniform distribution, where $p(\mathrm{s}) = 0$ while $\sum_{\mathbf{s}} p(\mathbf{s}) = 1$ as it is still a probability. This basically means all sources are equal probable.

<u>The Classifier</u>

The general properties of our model is currently based on allowing generation of new data from a sub-dimensional codevector s. If we assume the complete observed dataset consist of $C$ individual categories such that each datasample $\mathbf{x}$ has an attached label associating it to one of $C$ classes, then if a source vector $\mathbf{s}$ holding valid generative information it must also hold corresponding classifying information.

For classification problems we introduce a non-linear discriminant function as a mean to evaluate the relation of an observed datasample $\mathbf{x}$ to each of $C$ classes. One of the simplest discriminant functions defines a non-linear decision boundary or discriminant in multidimensional space and can be expressed on similar form as (4.7) by

$$\mathbf{v} = \mathbf{W}f(\mathbf{s}) \tag{4.12}$$

where $\mathbf{W}$ is the $C \times d+1$ dimensional weight matrix ($d+1$ due to the augmented threshold) and $f(\mathbf{s})$ is the corresponding $d$ dimensional non-linear source for the datasample $\mathbf{x}$. The row vectors of $\mathbf{W}$ can initially be interpreted as the normal-vectors defining the discriminative hyperplane in $d+1$ dimensional space for the linear case of $f(\mathbf{s}) = \mathbf{s}$, but since $f(\mathbf{s})$ is non-linear these discriminants are also non-linear. In addition these row vectors can also be seen as prototypes versions of $f(\mathbf{s})$ for each class. The $C$ dimensional output vector $\mathbf{v}$ then holds the quantitative relation to each class, where the input $\mathbf{s}$ should be labeled with the class $c$ with the highest element in $\mathbf{v} = \{v_c\}_{c=1}^{C}$.

A discriminant function on the form (4.12) also highlights the importance of the augmented threshold $\mathbf{b}$. With no bias the discriminative boundaries at $\mathbf{S} = 0$ intercept origo (assuming $f(0) = 0$) and thereby limits the discriminative capabilities of the function. By adding the threshold $\mathbf{b}$ we increase the flexibility of the classification. The parallel classification network can be combined with the generative network from the single layered architecture in figure 4.3 by augmenting the classifier as shown in figure 4.5. The figure also illustrates how an observed sample $\mathbf{x}$ is classified through the non-linear mapping of a corresponding source $\mathbf{s}$, this will be more clear later.

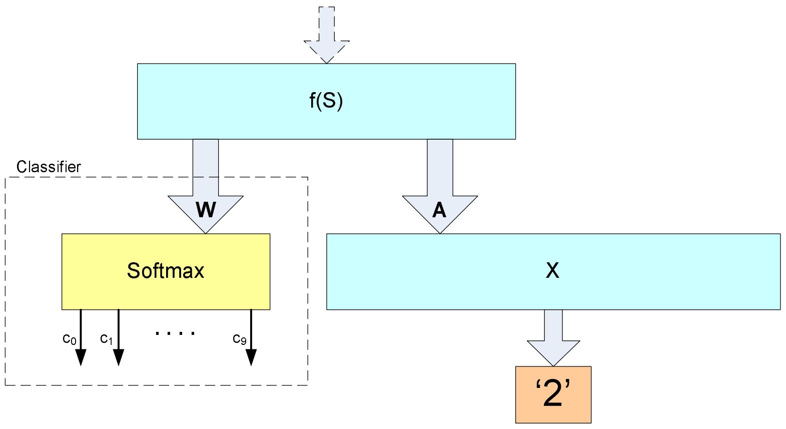

*Figure 4.5:* Architecture of the augmented classifying softmax function

Assume we want to classify a $d$-dimensional codevector $\mathbf{s}$ into $C$ classes, i.e. model the conditional posterior given by $y_c = p(c|\mathbf{s})$. In order to ensure the output $\mathbf{v}$ represents probabilities, i.e. $v_c \in [0;1]$ and $\sum_{c=1}^{C} v_c = 1$, we map the linear outputs through a normalized non-linear exponential function called *softmax* [4] to derive the posterior probabilities $y_c$ expressed as

$$y_c = p(c|\mathbf{s}) = \frac{\exp(v_c)}{\sum_{c'=1}^{C}\exp(v_{c'})} = \frac{\exp(\mathbf{w}_c f(\mathbf{s}))}{\sum_{c'=1}^{C}\exp(\mathbf{w}_{c'} f(\mathbf{s}))} \tag{4.13}$$

where $\mathbf{w}_c$ is the $c$'th row vector of the weight matrix $\mathbf{W}$. Using the softmax function to obtain class conditional probabilities has a built-in redundancy as $\sum_c y_c = 1$. This means we can model $C-1$ classes and infer the last class as $y_C = 1 - \sum_{c=1}^{C-1} y_c$. In reality this means a small reduction in computation time for the softmax function and has no effect on the classification performance.

In this context it is important to identify the difference between the generative network and the classification network in terms of modelling. The generative model given by (4.7) learns the features $\mathbf{A}$ and sources $\mathbf{s}$ unsupervised and seeks to represent the observed $\mathbf{x}$ such that the sources hold information maximizing reconstruction, which may be potentially all the elements in $\mathbf{s}$. In contrast the discriminative model in (4.7) is trained supervised and only learns parameters $\mathbf{W}$ and s by as much information as are required to specify the label of the observed $\mathbf{x}$. This means potentially only a few elements in s may be effective.

Hence the generative model can learn low-level features without requiring feedback from the labels and can thus learn many more parameters than the discriminative model. In addition the matrices $\mathbf{A}$ and $\mathbf{W}$ are also interpreted differently as mentioned despite the same form of decomposition in (4.7) and (4.12). By combining the generative and discriminative network we form a hybrid model trained both super- and unsupervised, which has the potential of improved classification performance compared to a stand-alone discriminative model.

With the introduction of the classifier network we can expand our probabilistic model in (4.9) to include dependency to the class labels by expressing the combined model by the joint distribution $p(\mathbf{x}, c)$ as

$$p(\mathbf{x}, c) = \int p(\mathbf{x}, \mathbf{s}, c)\ \mathrm{d}\mathbf{s} = \int p(\mathbf{x}|\mathbf{s})p(c|\mathbf{s})p(\mathbf{s})\ \mathrm{d}\mathbf{s} \tag{4.14}$$

Finally we introduce a set of non-linear mapping function inducing different constraints in relation to visual cognitive representation, defined next.

<u>Non-negative Constrained Mapping Function</u>

The non-linear mapping function $f(\mathbf{s})$ is essential for a deep network and can be formed with different constraints to extract features of interest. For the two first mapping functions we re-induce the constraint of non-negativity in order to extract sparse parts-based features with the motivation to obtain cognitive feature as for NMF in section 2.1.

For the non-linear decomposition in (4.7) we could initially define the non-linear mapping function $f(\mathbf{s})$ as a non-negative sigmoid function on the form $f(\mathbf{s}) = \frac{\exp(\mathbf{s})}{1+\exp(\mathbf{s})}$, shown left in figure 4.6. As the sigmoid function unfortunately also include the negative region of s and thereby intercepts in $f(0) = 1/2$ we indirectly introduce a bias to all sources of $1/2$, since we only model non-negative values of s, resulting of insufficient modelling of $\mathbf{X}$. To accommodate the sources are pre-mapped through a non-negative logistic function $\mathbf{s} \leftarrow \ln(\mathbf{s})$. By inserting s into $f(\mathbf{s})$, we finally obtain

$$f(\mathbf{s}) = \frac{\exp(\ln(\mathbf{s}))}{1+\exp(\ln(\mathbf{s}))} = \frac{\mathbf{s}}{1+\mathbf{s}} \tag{4.15}$$

The redefined $f(\mathbf{s})$ is now a non-linear function with $f(0) = 0$ and a dynamic range of [0;1] useful for non-negative mapping, as depicted right in figure 4.6.

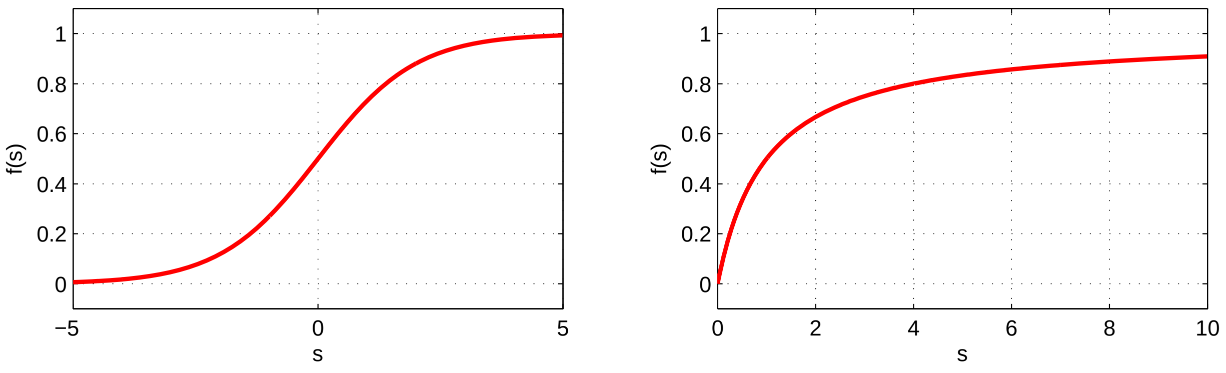

*Figure 4.6: Left*: The non-negative sigmoid function $f(s)$. *Right*: The non-negative mapping function in (4.15).

<u>Unconstrained Mapping Function</u>

A simple extension of the augmented non-linear matrix factorization in (4.15) is to negate the non-negativity constraint and allow full dynamic range of the sources s. This leads to potential modelling of the oriented Gabor-like filters in the primary visual cortex as discussed in section 1.2. Initially we could negate the extra pre-mapping function $s \leftarrow \ln(s)$ presented for the non-negative constraint to avoid negative values of s resulting in the initial function depicted left in figure 4.6 as our mapping function given by

$$f(\mathbf{s}) = \frac{\exp(\mathbf{s})}{1+\exp(\mathbf{s})} \tag{4.16}$$

As this function includes a bias at $f(0) = 1/2$, we enforce regulated sources to be located around $f(\mathbf{s}) = 1/2$ and not origo. This could however be accommodated by the threshold vector $\mathbf{b}$, but instead we introduce the full-range unbiased sigmoid tanh-function as an additional mapping function given by

$$f(\mathbf{s}) = \tanh(\mathbf{s}) \tag{4.17}$$

This non-linear mapping function has a dynamic range of [-1;1] and is symmetric around origo as shown in figure 4.7 making it suitable for mapping regulated sources.

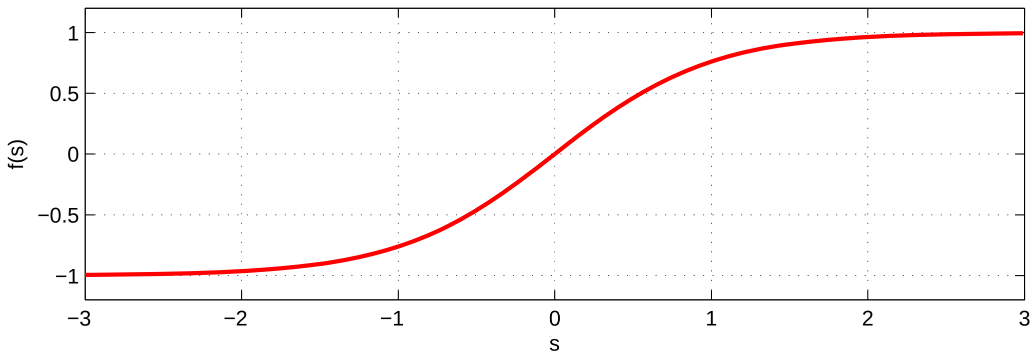

*Figure 4.7:* The unconstrained sigmoid mapping function $f(s)$.

Summary

With the motivation from deep belief nets by Hinton et al. [12] we have introduced a simpler generative deep network based on a continuous-valued linear model for each layer as opposed to the RBM proposed by Hinton et al.

The top-level prior $p(\mathbf{s})$ are modelling by the simple Laplace distribution leading to the absence of using complementary priors in contrast to using a joint distribution modelling an infinite undirected net. This means a reverse operation computing the posterior of the hidden units cannot be conducted by a simple propagation using transposed weights matrices.

## 4.3 Learning

In this section we describe how the model parameters are learned layer by layer. Our approach of learning is not based on a re-training or fine-tuning of the model in a second step in contrast to the deep belief model proposed by Hinton [12], where a greedy algorithm is only used for initializing parameters for a subsequent fine-tuning. This means our model will be easier to train, but may suffer from suboptimal solutions.

The network model given in (4.14) is parameterized by the codebook features $\mathbf{A}$ and the classification weights $\mathbf{W}$. To learn these parameters from an observed dataset $\mathbf{X} = \{\mathbf{x}_n\}_{n=1}^{N}$ we infer the parameters from maximizing the likelihood as described in 2.3.1. We derive a set of update rules initially for the generative model for the different mapping functions and afterwards for the augmented classification network defining the hybrid model.

### 4.3.1 Non-Linear Non-negative Matrix Factorization

For the generative model we infer the feature matrix $\mathbf{A}$ by maximum likelihood and thus define a cost function $E_{LS}$ as the negative log-likelihood of $p(\mathbf{x}|\mathbf{s})$. As derived in appendix A.6 and shown for regular NMF the negative log-likelihood for all samples leads to the least-squares estimate expressed as

$$E_{LS} = -\sum_{n=1}^{N} \ln p(\mathbf{x}_n|\mathbf{s}_n) = \frac{1}{2\sigma^2} \parallel \mathbf{X} - \mathbf{A}f(\mathbf{S}) \parallel^2 \tag{4.18}$$

The non-linearity induced in $f(\mathbf{s})$ provide the model with great flexibility to model highly non-linear structures, but also increases the probability of achieving an overfit, where the model (4.7) has adapted to the specific training samples instead of an underlying structure. In order to accommodate we induce a regulation of the codevectors $\mathbf{s}$, where the complexity of the model can be controlled. A classic approach for regulation is to augment the cost-function with an $L_1$- or $L_2$-norm weight-decay term. We therefore define our cost function as

$$E_{LS} = \frac{1}{2\sigma^2} \parallel \mathbf{X} - \mathbf{A}f(\mathbf{S}) \parallel^2 + \alpha|\mathbf{S}| \tag{4.19}$$

where $\alpha$ controls the degree of regulation. This approach ensures large values of $\mathbf{s}$ are given an additional penalty encouraging small or sparse sources located close to origo in the right figure of 4.6. The degree of regulation can further be adjusted by the reg.-factor $\alpha$, where $\alpha = 0$ corresponds to no regulation induced.

A large reg. factor will force the sources $\mathbf{s}$ to have more linear properties similar to (2.2) in the region close to origo of $f(\mathbf{s})$ reducing their non-linearity. In terms of optimizing the sources the regulation sets a lower bound of the cost-function in (4.19). In contrast this means a reg. factor too high may prevent the model from learning or adapting to an underlying structure of the data by enforcing a hard restriction on the sources. This is the classic *Bias/Variance* tradeoff and is discussed further shortly in appendix A.6.

Adding the regulation term and encouraging small values of $\mathbf{s}$ can also lead to a simple downscaling of the sources $\mathbf{s}$ and a corresponding upscaling of the columns of $\mathbf{A}$. In a strictly linear case, i.e. $\mathbf{X} = \mathbf{AS}$ a linear post-scaling of $\mathbf{A}$ and $\mathbf{S}$ are often included to avoid such case [22]. In our case with the non-linear $f(\mathbf{s})$ any scaling is not necessary due to the limited dynamic range of $\mathbf{s}$. Adding this type of regulation term to the cost function also corresponds to lowering the effective amount of parameters for the model [9].

The augmentation with the regulation term corresponds to including prior information of $\mathbf{s}$ derived from the negative log up to an additive constant of the Laplace distribution of the sources given in (4.11).

This means we are not inferring the parameters based on maximizing a likelihood, but on the posterior $p(\mathbf{S}|\mathbf{X})$ instead or by maximum a posteriori (MAP). From bayes in (2.13) maximizing the posterior corresponds to our cost function (4.19) defined as $-\ln p(\mathbf{X},\mathbf{S}) = -\ln p(\mathbf{X}|\mathbf{S})p(\mathbf{S})$. With this cost function we use the same steps as for regular NMF to derive a set of update rules for the sources, which becomes

$$\mathbf{S}_{r,n} \leftarrow \mathbf{S}_{r,n} - \Delta\frac{\partial E_{LS}}{\partial \mathbf{S}_{r,n}} = \mathbf{S}_{r,n} + \eta_{r,n}\left[\frac{1}{\sigma^2}\Big((\mathbf{A}^T\mathbf{X})_{r,n} - (\mathbf{A}^T\mathbf{A}f(\mathbf{S}))_{r,n}\Big)f'(\mathbf{S})_{r,n} - \alpha\,\mathrm{sign}(\mathbf{S})_{r,n}\right] \tag{4.20}$$

where $f'(\mathbf{S})$ denotes the 1st order derivate or gradient of the non-linear mapping function. By setting $\eta_{r,n} = \frac{\mathbf{S}_{r,n}}{1/\sigma^2\cdot(\mathbf{A}^T\mathbf{A}f(\mathbf{S}))_{r,n}f'(\mathbf{S})_{r,n}+\alpha}$ we can rewrite the update of $\mathbf{S}$ and similar for $\mathbf{A}$ into a set of multiplicative update rules given by

$$\mathbf{S}_{r,n} \leftarrow \mathbf{S}_{r,n}\frac{(\mathbf{A}^T\mathbf{X})_{r,n}f'(\mathbf{S})_{r,n}}{(\mathbf{A}^T\mathbf{A}f(\mathbf{S}))_{r,n}f'(\mathbf{S})_{r,n} + \alpha\cdot\sigma^2} \qquad \wedge \qquad \mathbf{A}_{m,r} \leftarrow \mathbf{A}_{m,r}\frac{(\mathbf{X}f(\mathbf{S})^T)_{m,r}}{(\mathbf{A}f(\mathbf{S})f(\mathbf{S})^T)_{m,r}} \tag{4.21}$$

This shows that the update equations for non-linear NMF are comparable with regular NMF forming a coordinate-descent "EM"-type algorithm with the non-linear mapping function $f(\mathbf{s})$ induced directly on the sources. The update for the sources $\mathbf{S}$ also reveal how the dependency to the gradient $f'(\mathbf{S})$ disappears if no regulation term is used, i.e. $\alpha = 0$. The convergence properties for (4.21) are still applicable as for regular NMF, refer to [18]. In addition the non-negativity as we achieved for regular NMF is also maintained for the non-linear NMF. In contrast this means the variables $\mathbf{X}$, $\mathbf{A}$ and $f(\mathbf{S})$ must all remain non-negative during the update.

In the practical implementation of the non-linear NMF, a step-size parameter is introduced similarly as for regular NMF in (2.9) to control the convergence speed. The threshold $\mathbf{b}$ is further maintained by only updating the first $d$ dimensions of the sources in (4.21), refer to appendix B.3 for the `MATLAB` code.

### 4.3.2 Non-linear Unconstrained Matrix Factorization

For the non-negative feature extraction in the previous section, the non-negative constraint provoke sparse codebook features in $\mathbf{A}$, which can be interpreted as cognitive (this will be more clear in later simulations). By disregarding the non-negative constraint the codebook elements are given max. freedom in terms of dynamic range and are hence not encouraged being sparse or parts-based as desired. To enforce sparse and cognitive features we augment the cost-function (4.19) with an additional regulation term for $\mathbf{A}$ by

$$E_{LS} = \frac{1}{2\sigma^2} \parallel \mathbf{X} - \mathbf{A}f(\mathbf{S}) \parallel^2 + \alpha|\mathbf{S}| + \beta|\mathbf{A}| \tag{4.22}$$

where $\beta$ adjusts the regulation of the codebook features in $\mathbf{A}$. This approach ensures large codebook features are penalized encouraging sparse features similar as for regulation of $\mathbf{s}$, where the degree of regulation is controlled by $\beta$. As for regulation of the sources, this approach sets a lower bound of the cost (4.22). Again this means a regulation factor which is too high may prohibit the features from learning or adapting to an underlying structure of the data by enforcing a hard restriction on the features (Bias/Variance tradeoff). Adding regulation of $\mathbf{A}$ to the cost function corresponds to including prior information of the codebook and can also derived from the negative log of the Laplace distribution as given in (4.11) for the sources. With now 2 different independent regulations care must be taken not to enforce a high level of regulation simultaneously as this may lead to a high lower-bound of the cost-function resulting in poor learning.

With these two choices of non-linear mapping functions (4.16) and (4.17), we can formulate the additive update equations from the cost function in (4.22) with the 1-norm sparsity terms through the same steps as for regular NMF to obtain (4.20), written here for convenience

$$\mathbf{S}_{r,n} \leftarrow \mathbf{S}_{r,n} - \Delta\frac{\partial E_{LS}}{\partial \mathbf{S}_{r,n}} = \mathbf{S}_{r,n} + \eta\left[\frac{1}{\sigma^2}\Big((\mathbf{A}^T\mathbf{X})_{r,n} - (\mathbf{A}^T\mathbf{A}f(\mathbf{S}))_{r,n}\Big)f'(\mathbf{S})_{r,n} - \alpha\ \mathrm{sign}(\mathbf{S})_{r,n}\right] \tag{4.23}$$

$$\mathbf{A}_{m,r} \leftarrow \mathbf{A}_{m,r} - \Delta\frac{\partial E_{LS}}{\partial \mathbf{A}_{m,r}} = \mathbf{A}_{m,r} + \zeta\left[\frac{1}{\sigma^2}\Big((\mathbf{X}f(\mathbf{S})^T)_{m,r} - (\mathbf{A}f(\mathbf{S})f(\mathbf{S})^T)_{m,r}\Big) - \beta\ \mathrm{sign}(\mathbf{A})_{m,r}\right] \tag{4.24}$$

where $f'(\mathbf{S})$ is the derivate or gradient of the non-linear mapping function. For $f(\mathbf{s})$ given in (4.16) the gradient is

$$f'(\mathbf{s}) = \frac{\partial}{\partial \mathbf{s}}\left[\frac{\exp(\mathbf{s})}{1+\exp(\mathbf{s})}\right] = \frac{1}{1+\exp(\mathbf{s})}\cdot\exp(\mathbf{s}) - \frac{\exp(\mathbf{s})}{\big(1+\exp(\mathbf{s})\big)^2}\cdot\exp(\mathbf{s}) = \frac{\exp(\mathbf{s})}{1+\exp(\mathbf{s})} - \left(\frac{\exp(\mathbf{s})}{1+\exp(\mathbf{s})}\right)^2 \tag{4.25}$$

$$= f(\mathbf{s}) - f^2(\mathbf{s}) \tag{4.26}$$

and for $f(\mathbf{s})$ given in (4.17) the gradient $f'(\mathbf{s}) = 1 - \tanh^2 = 1 - f(\mathbf{s})^2$. As the decomposition is unconstrained there is no need to adopt multiplicative update rules. This means the features in $\mathbf{A}$ (i.e. column vectors) can now hold negative elements as well leading to an increased modelling flexibility. The convergence of the cost-function to at least a local minimum is further ensured provided that the steps taken in both $\mathbf{A}$- and $\mathbf{S}$-space remain sufficiently small. To control the convergence rate both stepsize parameters $\eta$ and $\zeta$ are adjusted dynamically during iterations similar as for regular NMF. The threshold $\mathbf{b}$ is further maintained by only updating the $d$ first dimensions of the sources s in (4.23), refer to appendix B.3 for the `MATLAB` code. Since the non-linear mapping function $f(\mathbf{s})$ has dynamic range of [0;1], scaling during iterations can be omitted.

### 4.3.3 The Classifier Network

To infer the weight matrix $\mathbf{W}$ we conduct supervised learning using a $C$ dimensional binary target vector $\mathbf{t} = \{t_c\}_{c=1}^{C}$, where $t_c = 1$ if the codevector $\mathbf{s}$ belong to the $c$'th class, e.g. for $c = 3$, we get $\mathbf{t} = [\,0\;0\;1\;0\;\ldots\;0\,]$.

Initially we could learn $\mathbf{W}$ by optimizing a cost function based on the squared error expressed as $E_{sq} = \frac{1}{2}\sum_{n=1}^{N}\sum_{c=1}^{C}(y_c^{(n)} - t_c^{(n)})^2$, where the superscript denotes the n'th sample. This type of error function is typically based on the Gaussian noise hypothesis (refer to appendix A.6) as we assumed for the generative model. Since the targets $t_c$ are of discrete value in our in classification problem this error function is not appropriate. This is also evident when training classifying neural networks, where the squared-error function will give almost equal penalty for different degrees of misclassification, due to the saturating effect of the non-linearity of the activation function [4].

Instead we seek to maximize the likelihood of class label $t_c$ being generated from classifier output $y_c$, i.e. on vector form $p(\mathbf{t}|\mathbf{y})$ expressed as

$$p(\mathbf{t}|\mathbf{y}) = \prod_{c=1}^{C} y_c^{t_c} \tag{4.27}$$

We then define the cross-entropy cost-function $E_{EN}$ as the negative log-likelihood of (4.27) for all samples, i.e.

$$E_{EN} = -\sum_{n=1}^{N} \ln p(\mathbf{t}|\mathbf{y}) = -\sum_{n=1}^{N}\sum_{c=1}^{C} t_c^{(n)} \ln y_c^{(n)} \tag{4.28}$$

This cost function has the appealing property of penalizing large misclassifications harder and is thus appropriate to use in training a classifying network. An additional regulation term based on the $L_1$-norm is augmented to the cost function as a adjustable tool to control potential overfit. This corresponds to including prior information of the weights and is again derived from the negative log of the Laplace distribution as given in (4.11) for the sources. Thus to obtain the final expression for the cost function we insert the softmax function from (4.13) into (4.28) and get

$$E_{EN} = -\sum_{n=1}^{N}\left[\sum_{c=1}^{C} t_c^{(n)}\mathbf{w}_c f(\mathbf{s}_n) - \sum_{c=1}^{C} t_c^{(n)} \ln \sum_{c'=1}^{C} \exp\Big(\mathbf{w}_{c'} f(\mathbf{s}_n)\Big)\right] + \gamma|\mathbf{W}| \tag{4.29}$$

$$E_{EN} = -\sum_{n=1}^{N}\left[\sum_{c=1}^{C} t_c^{(n)}\mathbf{w}_c f(\mathbf{s}_n) - \ln \sum_{c'=1}^{C} \exp\Big(\mathbf{w}_{c'} f(\mathbf{s}_n)\Big)\right] + \gamma|\mathbf{W}| \tag{4.30}$$

where $\gamma$ is used to adjust to degree of regulation. By regulation we encourage smaller weights at the linear region of the softmax function and thereby achieve more linear discriminants leading to less complex modelling. With an appropriate regulation this approach can help prevent model overfitting, but choosing a regulation too strong may result in a trained model suffering from a bias. This will also be evident in later simulations.

To find the optimal weight matrix $\mathbf{W}$ we minimize the error by gradient descent and express the update of the individual row vectors $\mathbf{w}_c$ from the cost function (4.30)

$$\mathbf{w}_c \leftarrow \mathbf{w}_c - \Delta\frac{\partial E_{EN}}{\partial \mathbf{w}_c} = \mathbf{w}_c + \eta\left[\sum_{n=1}^{N}\left(t_c^{(n)} f(\mathbf{s}_n)^T - \frac{\exp(\mathbf{w}_c f(\mathbf{s}_n))}{\sum_{c'=1}^{C}\exp(\mathbf{w}_{c'} f(\mathbf{s}_n))} f(\mathbf{s}_n)^T\right) - \gamma\ \mathrm{sign}(\mathbf{w}_c)\right] \tag{4.31}$$

$$= \mathbf{w}_c + \eta\left[\sum_{n=1}^{N}\left(t_c^{(n)} f(\mathbf{s}_n)^T - y_c^{(n)} f(\mathbf{s}_n)^T\right) - \gamma\ \mathrm{sign}(\mathbf{w}_c)\right] \tag{4.32}$$

where the target $t_c$ in the first term inside the summation of the gradient ensures this term is only applied for the true class $c$. All other classes are updated with a weighted source vector $f(\mathbf{s})$.

With the introduction of the classifying function to the structure of the network the optimization of the non-linear codevectors $\mathbf{S}$ then also become dependent on the parameters of the classifier, $\mathbf{W}$ and not only the codebook matrix $\mathbf{A}$. This means the optimization equations for the non-negative constraint in (4.20) and the unconstrained in (4.23) must be updated to include dependency to the weight matrix $\mathbf{W}$.

This is achieved by defining a new complete cost function derived as the negative log of the joint distribution $p(\mathbf{x}, \mathbf{s}, c)$ derived from (4.14) omitting parameter dependencies. Hence from (4.22) and (4.30) the total cost function can be expressed as

$$E = E_{LS} + E_{EN} \tag{4.33}$$

$$E = \frac{1}{2\sigma^2} \parallel \mathbf{X} - \mathbf{AS} \parallel^2 + \alpha|\mathbf{S}| + \beta|\mathbf{A}| - \sum_{n=1}^{N} \left[ \sum_{c=1}^{C} t_c^{(n)} \mathbf{w}_c f(\mathbf{s}_n) - \ln \sum_{c'=1}^{C} \exp\Big(\mathbf{w}_{c'} f(\mathbf{s}_n)\Big) \right] + \gamma|\mathbf{W}| \tag{4.34}$$

Introducing the classifier to the network means the sources s must now also hold classifying information aside from reconstruction. The variance $\sigma^2$ can be seen as a balance parameter used to regulate the influence of $E_{LS}$ and $E_{EN}$ or the weight of class. or reconstruction information in the sources. For large $\sigma^2$ the classification cost is given higher influence risking overfit and for small $\sigma^2$ the reconstruction cost is favored risking leaving no influence to classification. A tradeoff must therefore be found as an optimal regulation. The two reg. terms $\alpha$ and $\beta$ are also affected by the balance parameter $\sigma^2$, but can easily be compensated if necessary as they are linear dependent.

The update equation for the codevectors $\mathbf{S}$ with softmax can now be found by deriving the gradient of (4.30) expressed as

$$\frac{\partial E_{EN}}{\partial \mathbf{S}_{r,n}} = -\Big(\sum_c t_c^{(n)} \mathbf{w}_c\Big)_r^T f'(\mathbf{S})_{r,n} + \left(\sum_c \frac{\exp\left(\mathbf{w}_c f(\mathbf{s}_n)\right)}{\sum_{c'} \exp\left(\mathbf{w}_{c'} f(\mathbf{s}_n)\right)} \mathbf{w}_c\right)_r^T f'(\mathbf{S})_{r,n} \tag{4.35}$$

$$= -\Big(\sum_c t_c^{(n)} \mathbf{w}_c\Big)_r^T f'(\mathbf{S})_{r,n} + \Big(\sum_c y_c^{(n)} \mathbf{w}_c\Big)_r^T f'(\mathbf{S})_{r,n} \tag{4.36}$$

$$= \Big(\sum_c \big(y_c^{(n)} - t_c^{(n)}\big) \mathbf{w}_c\Big)_r^T f'(\mathbf{S})_{r,n} \tag{4.37}$$

where $r$ denotes the index for the elementwise multiplications. This shows how the gradient becomes zero when the target $t_c$ matches the softmax output $y_c$, as expected. Based on our derivation of the update of s earlier in (4.23), we can express the update equation for optimizing the unconstrained sources as

$$\mathbf{S}_{r,n} \leftarrow \mathbf{S}_{r,n} - \Delta \frac{\partial E}{\partial \mathbf{S}_{r,n}} = \mathbf{S}_{r,n} + \eta_{r,n} \Bigg[ \frac{1}{\sigma^2} \Big( (\mathbf{A}^T \mathbf{X})_{r,n} - (\mathbf{A}^T \mathbf{A} f(\mathbf{S}))_{r,n} \Big) f'(\mathbf{S})_{r,n} - \alpha \operatorname{sign}(\mathbf{S})_{r,n} + \Big(\sum_c \big(t_c^{(n)} - y_c^{(n)}\big) \mathbf{w}_c\Big)_r^T f'(\mathbf{S})_{r,n} \Bigg] \tag{4.38}$$

This equation and (4.32) form the update rules for unconstrained optimization of the sources s with a softmax classifier. For the non-negative constraint on the sources we derive an update equation by setting $\eta_{r,n} = \frac{\mathbf{S}_{r,n}}{1/\sigma^2 \cdot (\mathbf{A}^T \mathbf{A} f(\mathbf{S}))_{r,n} f'(\mathbf{S})_{r,n} + \alpha + (\sum_c y_c^{(n)} \mathbf{w}_c)_r f'(\mathbf{S})_{r,n}}$ and rewrite the update of $\mathbf{S}_{r,n}$ to

$$\mathbf{S}_{r,n} \leftarrow \mathbf{S}_{r,n} \frac{1/\sigma^2 \cdot (\mathbf{A}^T \mathbf{X})_{r,n} + \big(\sum_c t_c^{(n)} \mathbf{w}_c\big)_r}{1/\sigma^2 \cdot (\mathbf{A}^T \mathbf{A} f(\mathbf{S}))_{r,n} + \big(\sum_c y_c^{(n)} \mathbf{w}_c\big)_r + \alpha / f'(\mathbf{S})_{r,n}} \tag{4.39}$$

From both (4.38) and (4.39) we see how the balance parameter $\sigma^2$ regulate the influence of the reconstruction and classification gradient during the update of s. Eventhough the update equation (4.39) is strictly multiplicative, we can only ensure a non-negative update provided the elements in the codebook matrix $\mathbf{A}$ and the weight matrix $\mathbf{W}$ are all non-negative. For the features in $\mathbf{A}$ we employ similar multiplicative update as given in (4.21), but for the weights in $\mathbf{W}$ non-negativity cannot be guaranteed from the additive updates in (4.32). We therefore also constrain the weights to be non-negative and derive a multiplicative update of the rows in $\mathbf{W}$ by setting $\eta_r = \frac{(\mathbf{w}_c)_r}{\left(\sum_{n=1}^N y_c^{(n)} f(\mathbf{s}_n)^T\right)_{r,n} + \gamma}$ and insert into (4.32)

$$\mathbf{w}_{c,r} \leftarrow \mathbf{w}_{c,r} \frac{\left(\sum_{n=1}^N t_c^{(n)} f(\mathbf{s}_n)^T\right)_r}{\left(\sum_{n=1}^N y_c^{(n)} f(\mathbf{s}_n)^T + \gamma\right)_r} \tag{4.40}$$

where $\gamma$ in this case is a $d+1$ dimensional vector. The nominator shows how only the true-class samples contributes to the update of W enforced by the binary $t_c$. In addition it is also evident to see how gradient become zero when the output $y_c$ equals the desired target $t_c$ assuming no regulation $\gamma = 0$. This expression and (4.39) constitute the update rules for the classifying network for non-negative constrained optimization.

Augmenting the classifier has given a new set of update rules for $\mathbf{A}$, $\mathbf{S}$ and $\mathbf{W}$, which form an iterative co-ordinate descent "EM"-type algorithm comparable with regular NMF. For the multiplicative update rules the variables must all remain non-negative during the update to uphold the non-negativity constraint and the convergence properties are still applicable as for regular NMF, refer to [17].

In the practical implementation a step-size parameter is introduced as before to control the convergence speed. The threshold $\mathbf{b}$ is further maintained by only updating the $d$ first dimensions of the sources $\mathbf{s}$ in (4.38) and (4.39), refer to appendix B.3 for the MATLAB code.

Network Modules

In building the network we have introduced different decompositions in sections 4.3.1 & 4.3.2 with and without regulation and a parallel classifying network. To summarize we can consider each layer as a module parameterized by $\Theta = \{d, \alpha, \beta, \gamma, \sigma^2\}$ either individually or as a common set. This leaves a vast set of combinations when forming the building blocks or modules for the multilayered model. The general architecture of our deep net model can be visualized as in figure 4.8.

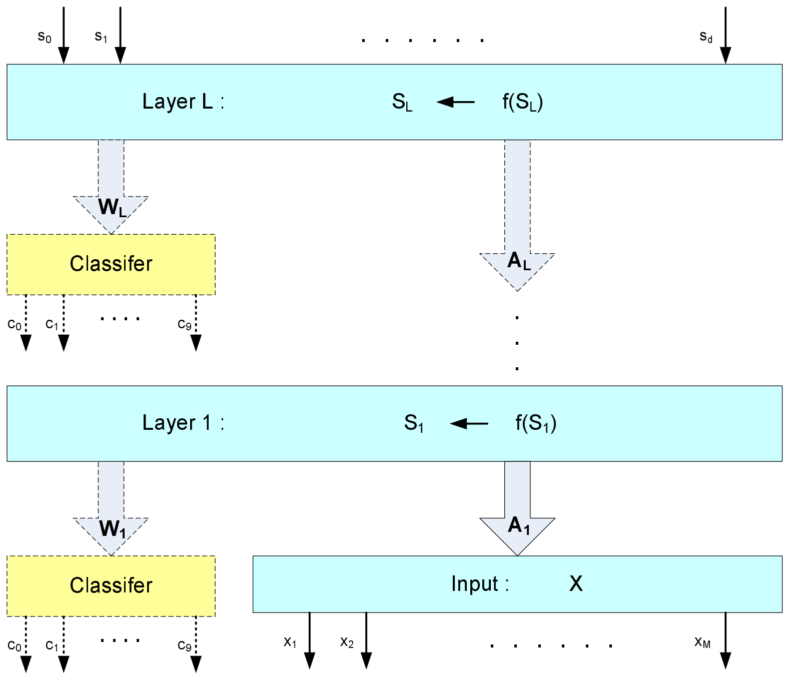

*Figure 4.8:* Architecture of the multilayered deep network model with $L$ layers.

Some of the properties of the modules are however mutually exclusive. From the non-negative update rules in (4.21) we see that enforcing the non-negative constraint on for instance $\mathbf{A}$ can only be upheld if the non-linear sources $f(\mathbf{S})$ are also non-negative and vice versa. We have also argued the innecessesity of regulating for sparse codebook features in $\mathbf{A}$ for the non-negative constraint. Hence using the non-negative constraint means all parameters must share the same constraint in our case. In contrast the unconstrained decomposition are not restricted to a specific dynamic range of its parameters, as can be seen from the update rules (4.24), (4.32) and (4.38).

These approaches are assuming a single training session, where the $\mathbf{A}$, $\mathbf{S}$ and $\mathbf{W}$ are optimized in one round. As mentioned this sets a limitation for the non-negative approach, where in particular the classification weight matrix $\mathbf{W}$ must remain non-negative. By training the model in two sessions we can achieve unconstrained weights $\mathbf{W}$ leading to potentially improved classification. In the first round the network is trained with the non-negativity constraint without the classification part attached meaning classification information is not taken into account in training the sources $\mathbf{S}$. In a second step the classification weights $\mathbf{W}$ are optimized supervised and unconstrained with the non-negative sources $\mathbf{S}$ from the first session. This dual session approach hence leads to non-negative sources $\mathbf{S}$ and unconstrained weights $\mathbf{W}$.

To summarize table 4.2 lists the set of module combinations.

| Session | $\mathbf{A}$ | $\mathbf{s}$ | $\mathbf{W}$ | $f(\mathbf{s})$ | $\alpha$ | $\beta$ | $\gamma$ | $\sigma^2$ |
|---|---|---|---|---|---|---|---|---|
| Case 1 | Non-Neg. | Non-Neg. | Non-Neg. | nn $\rightarrow$ nn | ☒ | ☐ | ☒ | ☒ |
| Case 2 | Uncon. | Uncon. | Uncon. | un $\rightarrow$ un | ☒ | ☒ | ☒ | ☒ |
| Case 3 | Non-Neg. | Uncon. | Non-Neg. | un $\rightarrow$ nn | ☒ | ☐ | ☒ | ☒ |
| Case 4 (dual) | Non-Neg. | Non-neg | Uncon | nn $\rightarrow$ nn | ☒ | ☐ | ☒ | ☐ |

*Table 4.2:* Module combinations.

## 4.4 Classification of data

With our trained classifier we can classify new data $\mathbf{x}$ by evaluating the posterior $p(c|\mathbf{x})$ for each class $c$ and choose the largest. Thus to find $p(c|\mathbf{x})$ we can expand

$$p(c|\mathbf{x}) = \frac{p(\mathbf{x}, c)}{\sum_c p(\mathbf{x}, c)p(c)} \tag{4.41}$$

where the denominator can be neglected for classification tasks as it is class independent. As discussed earlier our simplified deep network suffered from the ability to sample from the posterior $p(\mathbf{s}^{l+1}|\mathbf{s}^l)$ as we do not have complementary priors and thus data cannot be classified by simple propagation up through the layers. As the joint distribution $p(\mathbf{x}, c)$ as given in (4.14) is difficult to compute due to the marginalization over the sources, we approximate $p(\mathbf{x}, c)$ by estimating an optimal source $\mathbf{s}_x$ from the likelihood $p(\mathbf{x}|\mathbf{s})$ as only this depends on the data $\mathbf{x}$ in (4.14), hence

$$\mathbf{s}_x = \underset{\mathbf{s}}{\operatorname{argmax}}\ p(\mathbf{x}|\mathbf{s})p(\mathbf{s}) \tag{4.42}$$

The joint distribution can now be written as $p(\mathbf{x}, c) \approx p(\mathbf{x}|\mathbf{s}_x)p(c|\mathbf{s}_x)p(\mathbf{s}_x)$ and thus using $\mathbf{s}_x$ for classification corresponds to evaluating the conditional posterior $p(c|\mathbf{x}, \mathbf{s}_x)$, which can be expanded as

$$p(c|\mathbf{x}, \mathbf{s}_x) = \frac{p(\mathbf{x}, \mathbf{s}_x, c)}{p(\mathbf{x}, \mathbf{s}_x)} = \frac{p(\mathbf{x}|\mathbf{s}_x)p(c|\mathbf{s}_x)p(\mathbf{s}_x)}{\sum_c p(\mathbf{x}|\mathbf{s}_x)p(c|\mathbf{s}_x)p(\mathbf{s}_x)} = \frac{p(\mathbf{x}|\mathbf{s}_x)p(c|\mathbf{s}_x)p(\mathbf{s}_x)}{p(\mathbf{x}|\mathbf{s}_x)p(\mathbf{s}_x)} \tag{4.43}$$

$$= p(c|\mathbf{s}_x) \tag{4.44}$$

Hence the class $c$ is independent of $\mathbf{x}$ once we have estimated $\mathbf{s}_x$ as thus leads to a new representation of $\mathbf{x}$ used for evaluating the class probability $p(c|\mathbf{s}_x)$.

The practical approach of classifying an observed datasample $\mathbf{x}$ is to infer the corresponding source $\mathbf{s}_x$ with a constant codebook matrix $\mathbf{A}$. Initially we could apply the pseudo-inverse of $\mathbf{A}$ to obtain $\mathbf{s}_x$, i.e. $\mathbf{s}_x = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{x}$ derived from the least-squares cost function in (4.22). However this is only applicable for the strictly linear case where $f(\mathbf{s}) = \mathbf{s}$ and hence due to the non-linearity of $f(\mathbf{s})$ this approach is not feasible.

Instead we infer $\mathbf{s}_x$ by initiating an iterative learning process using either (4.21) or (4.23). From the corresponding source $\mathbf{s}_x$ the class can easily be inferred by propagating through the classifier using (4.13) and thereby obtain and evaluate $p(c|\mathbf{s}_x)$. For classification the same `MATLAB` code used for learning the model parameters can be used and can be found in appendix B.3.

## 4.5 Generation of data

In generating new class-conditional data $\mathbf{x}$ with the same statistical properties as the training examples the objective is to generate valid sample $\mathbf{x}$ with the correct class label by extracting samples from the posterior $p(\mathbf{x}|c)$, which can be expanded into

$$p(\mathbf{x}|c) = \frac{p(\mathbf{x}, c)}{\sum_x p(c|\mathbf{x})p(\mathbf{x})} \tag{4.45}$$

where the denominator can be neglected for generation tasks as it is independent of $\mathbf{x}$.

In deep belief net by Hinton et al. the top level associative memory modelling an infinite directed model with tied weights is used to generate top-level sources by alternating Gibbs sampling while clamping a class label. This means the sources are extracted from a relatively complex prior distribution $p(\mathbf{h}^L, \mathbf{h}^{L-1}, \mathbf{c})$ in contrast to our crude approximation using the Laplace distribution for our priors $p(\mathbf{s})$ given in (4.11).

As our model requires computing the inconvenient $p(\mathbf{x}, c)$ we approximate the distribution by estimating an optimal source $\mathbf{s}_c$ from the conditional probability $p(\mathbf{s}, c)$

$$\mathbf{s}_c = \underset{\mathbf{s}}{\operatorname{argmax}}\ p(c|\mathbf{s})p(\mathbf{s}) \tag{4.46}$$

The joint distribution can now be expressed as $p(\mathbf{x}, c) \approx p(\mathbf{x}|\mathbf{s}_c)p(c|\mathbf{s}_c)p(\mathbf{s}_c)$ and thus using $\mathbf{s}_c$ for generation corresponds to evaluating the conditional posterior $p(\mathbf{x}|\mathbf{s}_x, c)$, which can be expanded into

$$p(\mathbf{x}|\mathbf{s}_c, c) = \frac{p(\mathbf{x}, \mathbf{s}_c, c)}{p(\mathbf{s}_c), c} = \frac{p(\mathbf{x}|\mathbf{s}_c)p(c|\mathbf{s}_c)p(\mathbf{s}_c)}{\sum_{\mathbf{x}} p(\mathbf{x}|\mathbf{s}_c)p(c|\mathbf{s}_c)p(\mathbf{s}_c)} = \frac{p(\mathbf{x}|\mathbf{s}_c)p(c|\mathbf{s}_c)p(\mathbf{s}_c)}{p(\mathbf{x}|\mathbf{s}_c)p(\mathbf{s}_c)} \tag{4.47}$$

$$= p(\mathbf{x}|\mathbf{s}_c) \tag{4.48}$$

Hence $\mathbf{x}$ is independent of $c$ once we have estimated $\mathbf{s}_c$. We can now extract new data $\mathbf{x}$ from the conditional distribution $p(\mathbf{x}|\mathbf{s}_c)$ and thus to extract sources from $p(c, \mathbf{s})$ we derive a MAP estimate by minimizing a cost function defined as the negative log of $p(c, \mathbf{s})$ derived from (4.13) and (4.11), i.e.

$$E_{GEN} = -\ln p(c, \mathbf{s}) = -\ln p(c|\mathbf{s}) - \ln p(\mathbf{s}) \tag{4.49}$$

$$= -t_c \mathbf{w}_c f(\mathbf{s}) + \ln \sum_{c'=1}^{C} \exp\left(\mathbf{w}_{c'} f(\mathbf{s})\right) - \ln\frac{\alpha}{2} + \alpha|\mathbf{s}| \tag{4.50}$$

where $\mathbf{t}$ is the binary target vector defined earlier holding the desired class label to extract from. This cost function is basically the same derivation as in (4.30). Hence to generate labeled sources we minimize the error by gradient descent and express the update using use the same calculations from (4.37) and get

$$\mathbf{s}_r \leftarrow \mathbf{s}_r - \Delta\frac{\partial E_{GEN}}{\partial \mathbf{s}_r} = \mathbf{s}_r - \eta_r \left[ \Big( \sum_c (y_c - t_c)\mathbf{w}_c \Big)_r^T f'(\mathbf{s})_r + \alpha\ \text{sign}(\mathbf{s})_r \right] \tag{4.51}$$

This also shows how the class error $y_c - t_c$ is propagated back through the classification network by applying the weights and the 1st order derivative of the non-linear mapping function. The convergence rate is again controlled by dynamically adjusting the stepsize $\eta$ during iterations as for regular NMF. In order to avoid generating the same digit for the same class we initialize the sources randomly and use a stop criteria based on a threshold of $p(c|\mathbf{s})$, refer to the `MATLAB` code in appendix B.3.

One of the potential deficits of this approach in generating new data is that it is mainly based on the crude approximation of modelling the sources with a Laplace distribution in (4.11). Initially this means the quality of the data may be very poor if this distribution is far from the true. In addition if the respective model to generate from is optimal for no regulation of the sources, i.e. $\alpha = 0$ then $p(\mathbf{s})$ can be neglected and our generation is solely based on the classification weights $\mathbf{W}$. This will be more clear in our simulations. For the purpose of generation data the deep belief network of Hinton et al. has a great advantage in this respect due to its top-level associative memory as discussed earlier.

# 5. Deep Network Simulations

In section 4 we introduced the deep network model using on the decomposition $\mathbf{x} = \mathbf{A}f(\mathbf{s})$ as our non-linear generative model. Based on this decomposition we aim to extract sources $\mathbf{s}$, which holds sufficient information to generate and classify digits from the MNIST dataset. In this section we will present different model architectures and analyze their corresponding performance in terms of generation and classification. We will start the analysis with a single layered model to evaluate the impact and significance of the model parameters.

The network model types described in table 4.2 prove the vast combinations of building a model, but due to a limited timeframe and the unfortunate very time consuming simulations (up to 3 days pr. simulation), we limit our analysis to only one model type. The main analysis will therefore be based on the unconstrained model mainly due to its flexibility in modelling using all parameters and also for its potential of extracting features resembling receptive fields. A smaller analysis on the non-negative module will however also be given.

The `MATLAB` code used for the simulations can be found in appendix B.3.

## 5.1 The MNIST Dataset

Our simulations for the deep network model are still being based on the MNIST dataset of handwritten digits as presented in section 3.1. In order to evaluate the model performance in classification problems we include all digit classes from the dataset. For the full dataset of 60000 samples, this leads to very large amounts of data to analyze for the different layers in the model ($\sim$ 250Mb for $d = 500$ features pr. layer) and are very time consuming, we conduct our analysis on a smaller subset with only 10000 samples (1000 from each class) in both the training and test set. The same pre- and post-processing described in section 3.1 is still applicable for this analysis.

## 5.2 Single Layer Network

Before building the hierarchal architecture of our model, we give an analysis of the influence and impact of the model parameters $\Theta = \{d, \alpha, \beta, \gamma, \sigma^2\}$ and evaluate the output and performance of a single layer as depicted in figure 4.8 in terms of classification of digits. Afterwards a small analysis of the models generative properties will be conducted.

To start the analysis we train a single layer network with $d = 50$ features with parameters $\Theta = \{50, 0, 0, 0, 1\}$ and evaluate the extracted features based on both the non-negative (NN) and unconstrained (UN) decompositions presented in section 4.3.1 and 4.3.2. For stop criteria of the learning, we use $\Delta E = 1e-5$ and a max. of 1000 iterations. This set will be used henceforth unless otherwise noted. The decrease of the cost function both $E_{LS}$ and $E_{EN}$ can be seen in figure 5.1 for NN and UN features.

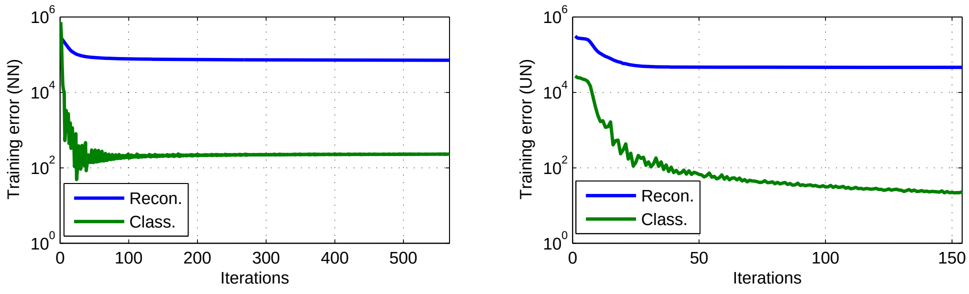

*Figure 5.1:* Convergence of training cost both reconstruction and classification for NN (left) & UN (right) features with parameters $\Theta = \{50, 0, 0, 0, 1\}$.

The sudden increase and small fluctuations of the classification test error is caused by the update of the sources $\mathbf{s}$ favoring the reconstruction error in this case. This is the battle between favoring either the reconstruction error (4.22) or the classification error (4.30). The parameter $\sigma^2$ serves exactly to adjust the balance between these updates and thus control the share of class. vs. recon. information in the sources. In addition the converged errors show how the UN model has achieved lower training error than the NN model for both classification and reconstruction. This indicates that the non-negative constraint model suffers from a bias as it prohibits learning the features properly or that the UN model has overfitted to the training data. The corresponding features are depicted in figures 5.2 and 5.3.

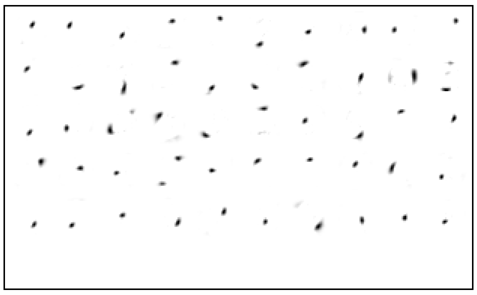
*Figure 5.2:* Extracted features for NN for $d = 50$ features without regulation, where the threshold patch is in the lowest left corner. Parameters $\Theta = \{50, 0, 0, 0, 1\}$.

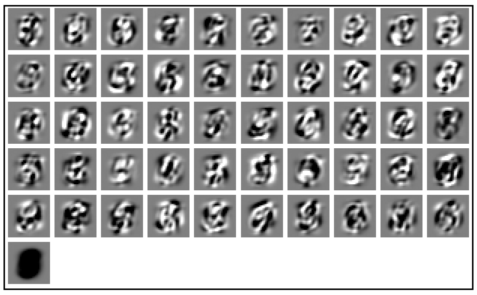
*Figure 5.3:* Extracted features for UN for $d = 50$ features without regulation, where the threshold patch is in the lowest left corner. Parameters $\Theta = \{50, 0, 0, 0, 1\}$.

The features for NN model have proven to be very sparse and localized and coincide with the prior definition of cognitive components and can therefore be considered as such. The threshold vector $\mathbf{b}$ in the lowest left corner of figure 5.2 is the zero vector (all elements are zero). As the NN model cannot subtract elements any non-zero threshold vector $\mathbf{b}$ must be part of all training digits. Hence a zero threshold vector indicates there are no average common pixels for the digits in the training set.

In contrast the features extracted by the UN model can be characterized as non-local coarse segments with small indications of Gabor-like structures and could be said to resemble hypercomplex cells (eventhough their definition is not quite clear), but can hardly be considered cognitive. In addition the threshold vector $\mathbf{b}$ has high contrast energy with a faint structure and is non-zero due to the ability to eliminate or subtract unwanted elements. Despite the features may not be very cognitive the inhibitory effect induced in the UN model is essential as seen from the error rates in figure 5.1 since it allows greater flexibility in modelling cognitive data.

### 5.2.1 Unconstrained Model

A detailed analysis of the impact of the regulations is made for the unconstrained model.

Size of hidden layer, $d$

The size of the network $d$ obviously has a major influence on the performance wrt. classification and reconstruction. To evaluate the impact the codebook features in $\mathbf{A}$ are shown in figure 5.4 for increasing network sizes (only a subset shown). The figure shows how the spatial frequency of the features is proportional with the network size as they gradually resemble noise without any clear structure.

A set of corresponding reconstructed training digits are shown in figure 5.5 illustrating the increasing quality of the generated data. In addition the grey background color becomes gradually whiter indicating that the negative elements in the generated digits fade out for larger networks. Considering the very noisy non-structural features for $d = 300$ (last row in figure 5.4) it is amazing how these features can in fact be used to generate digits of such quality as the last row in figure 5.5 depicts.

The gradually improving reconstruction of digits can also be seen from the corresponding decaying reconstruction testerror shown left in figure 5.6 given by $e_r = \frac{1}{2N} \parallel \mathbf{X} - \mathbf{A}_1\mathbf{S}_1 \parallel^2$. The decrease indicates the model suffers from a bias wrt. reconstruction.

The classification testerror is depicted in the middle illustration of figure 5.6 clearly revealing an optimal network size around $d = 50$. Larger networks overfit leading to the degraded testerror, which is also evident from

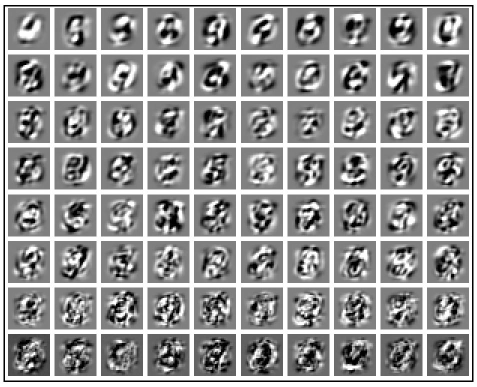

*Figure 5.4:* Extracted features for network sizes $d = \{20, 30, 50, 60, 80, 100, 200, 300\}$ (from top row) with parameters $\Theta = \{d, 0, 0, 0, 1\}$.

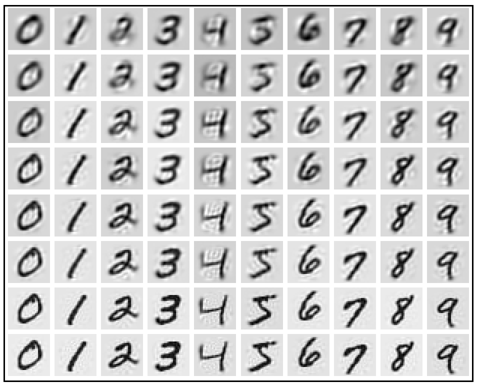

*Figure 5.5:* Reconstructed digits for different network sizes $d = \{20, 30, 50, 60, 80, 100, 200, 300\}$ (from top row) with parameters $\Theta = \{d, 0, 0, 0, 1\}$.

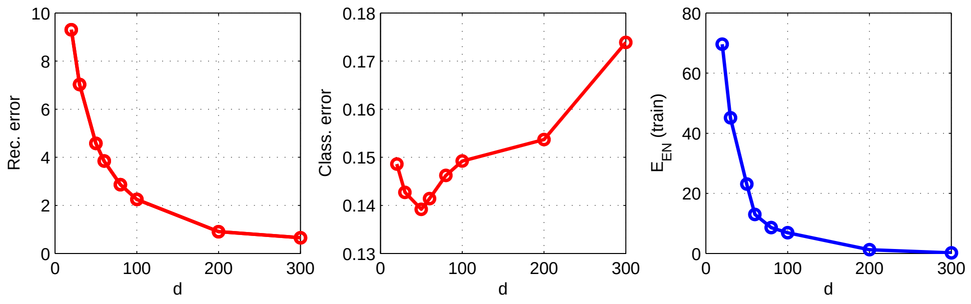

*Figure 5.6:* Performance for different sizes of hidden layers d with parameters $\Theta = \{d, 0, 0, 0, 1\}$. *Left:* Reconstruction error (test). *Mid:* Class. error (test). *Right:* Class. error (train).

the class. training error given in eq. (4.30) shown right in the figure, where the network adapts increasingly to the trainingset.

A subset of the corresponding misclassified digits for $d = 50$ hidden units are illustrated in figure 5.7. It can be seen how most classes hold misclassified digits, which are similar in structure to the true class. This is particularly evident for class 0, 1, 4 and 9, where for instance most digits are slim and narrow for class 1 and round and plump for class 0.

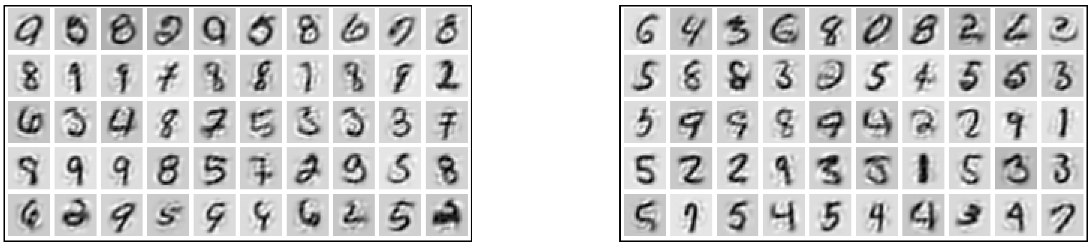

*Figure 5.7:* Subset of misclassified digits from model with parameters $\Theta = \{50, 0, 0, 0, 1\}$. Each row represents misclassified digits from each class starting from top left row of class zero.

To summarize this means the sources s fail to capture the complete generative structure of the trainingset, but have simultaneously adapted too strong to the classifying information in the trainingset.

### Regulation of sources ($\alpha$)

The overfit of the classification error can initially be controlled by regulating the sources s directly. By regulation we encourage smaller sources located in the linear region around origo at $f(\mathbf{s})$ (for the UN model) as shown in figure 5.8.

In addition regulation also encourage more sparse sources as shown in figure 5.9.

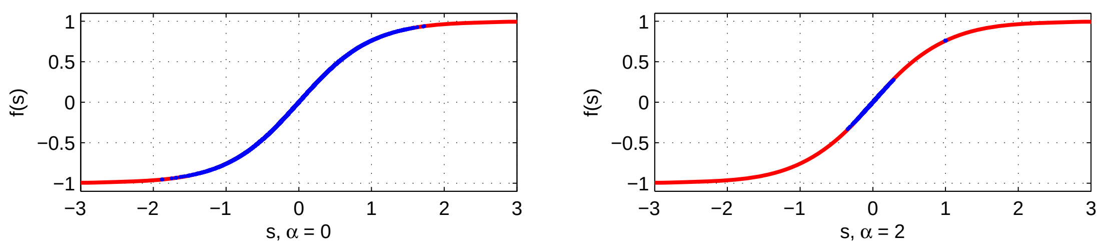

*Figure 5.8:* Mapping of sources into $f(s)$ for both no regulation (*left*) and strong regulation (*right*) with parameters $\Theta = \{500, \alpha, 0, 0, 1\}$.

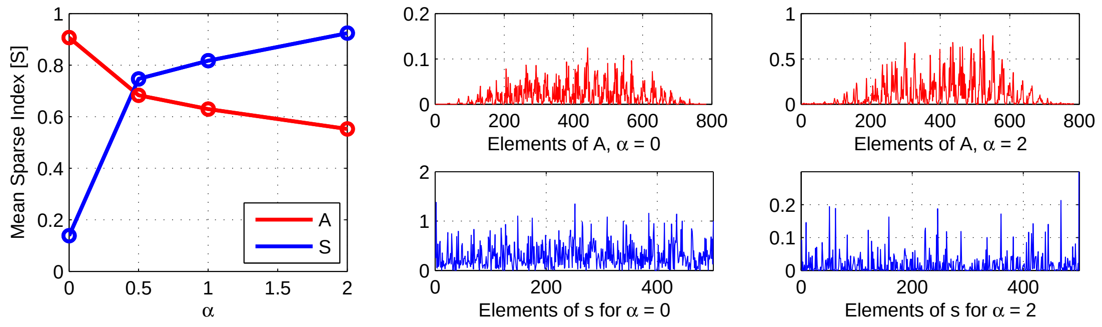

*Figure 5.9:* The mean sparsity index of $\mathbf{A}$ & $\mathbf{S}$ (*left*) and examples of corresponding features (*top*) and sources (*bottom*) with parameters $\Theta = \{500, \alpha, 0, 0, 1\}$ for no and strong regulation.

To achieve a more quantitative expression of the sparsity, we define a mean sparsity index (MSI) as mean of the amount of elements within a vector below a predefined threshold. The left figure depicts the mean sparse index with a threshold of $5\%$ for both the codebook features $\mathbf{A}$ and the sources $\mathbf{S}$. Initially it reveals how the sources become more sparse for stronger regulation, but since the mean sparse index is based on the data directly (i.e. not pre-normalized) low-energy sources will raise the MSI. Evaluating the actual elements of s (bottom figures) we see that the sources are more small than sparse. In addition the left illustration also show how the sparsity of codebook features $\mathbf{A}$ is inverse proportional to the sources. Achieving the smaller sources means the energy of the codebook features $\mathbf{A}$ correspondingly increases as a compensation to maintain low reconstruction error as can be seen in the top figures. This leads to more elements in $\mathbf{A}$ above the sparsity threshold leading to the decrease in sparsity.

Figure 5.10 illustrates the performance for different degrees of regulation for a large network with $d = 500$ hidden variables showing both worse classification and reconstruction errors.

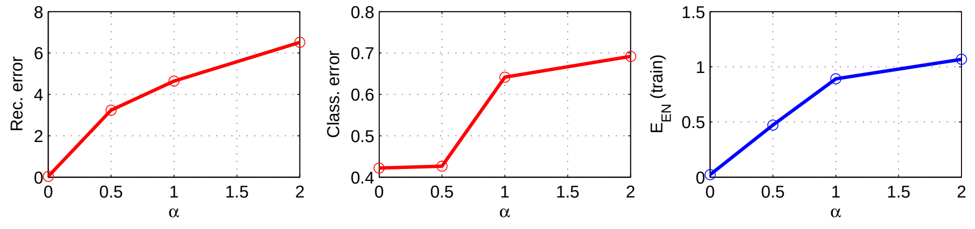

*Figure 5.10:* Performance for different regulations of $\alpha$ with parameters $\Theta = \{500, \alpha, 0, 0, 1\}$. *Left:* Reconstruction error (test). *Mid:* Class. error (test). *Right:* Class. error (train).

By regulating, the sources become poorly trained failing to capture the underlying generative structure in the trainingset, which forms sources not holding sufficient generative information (left figure). This leads to worse classification simply due to poorly trained sources and thus regulation of the sources does not attenuate the overfit of class. information in s.

Balance Parameter $\sigma^2$

The tradeoff between optimizing the sources s for either classification or reconstruction can be regulated by the balance parameter $\sigma^2$ in eq. (4.34). To illustrate figure 5.11 shows the effect of regulating $\sigma^2$ for a small network with $d = 50$ features.

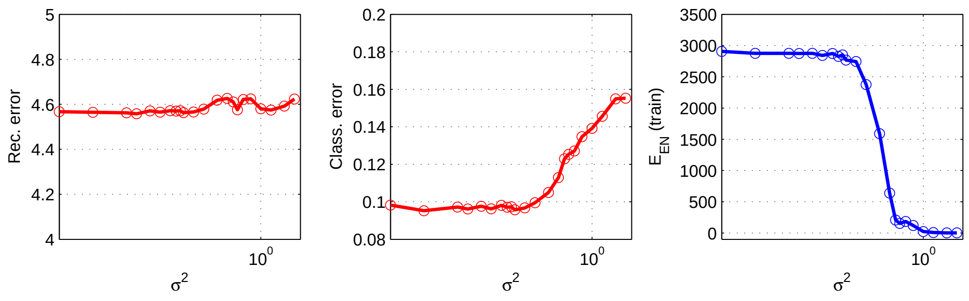

*Figure 5.11:* Performance for different values of $\sigma^2 = \{1e-6, \ldots, 10\}$ with parameters $\Theta = \{50, 0, 0, 0, \sigma^2\}$. *Left:* Reconstruction error (test). *Mid:* Class. error (test). *Right:* Class. error (train).

Initially it is evident to see that the reconstruction error in the left figure is almost unaffected for different balances of $\sigma^2$. More importantly the middle and right figures clearly reveal an overfit of the sources $\mathbf{s}$ to classification information for higher values of $\sigma^2$. This can be seen from the high test-errors and correspondingly low trainingerror, where the model has adapted to closely to the training set.

For lower balances of $\sigma^2$ the classification test error is relatively stable. However for extremely low $\sigma^2$, where the sources are not optimized with any classification information we expect the test error to increase a bit. A preliminary simulation in training sources without the classification network and afterwards optimizing the class. weights $\mathbf{W}$ revealed a testerror of 16.7%, which is a modest increase. This proves that including supervised classification information when training the sources is vital for a good class. performance. This important relation has also been discovered by Bengio et. al. [2].

In addition the middle figure also indicates a threshold at appr. $\sigma^2 = 1e-2$, where the classification test error is sufficiently regulated to avoid overfit. As always larger networks has a greater propensity to overfit due to the increased modelling flexibility and thus need to be regulated harder. For the balance parameter $\sigma^2$ the same regulative impact can be identified for such larger network of $d = 200$ as illustrated in figure 5.12.

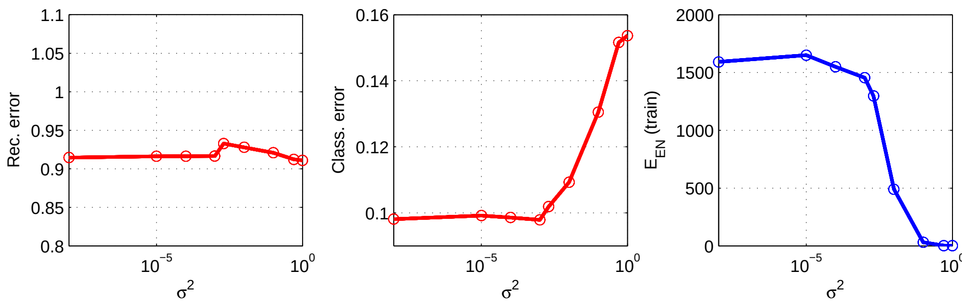

*Figure 5.12:* Performance for different values of $\sigma^2$ with parameters $\Theta = \{200, 0, 0, 0, \sigma^2\}$. *Left:* Reconstruction error (test). *Mid:* Class. error (test). *Right:* Class. error (train).

The same classification overfitting can be seen, where the threshold has decreased to appr. $\sigma^2 = 1e-3$ proving the need for stronger regulation for larger networks.

For a more clear analysis of how the classification information is stored in the sources we can analyze the weights in $\mathbf{W}$ and the sources $\mathbf{s}$ wrt. emphasized elements. For most classes the weights are relatively flat, i.e. no distinct elements (not shown). However in figure 5.13 the most prevalent classes are illustrated.

Initially the bottom left figure depicts the source elements for all 1 digits. This shows a few distinct elements, which are characteristic of the 1 digits, but since these elements may be important for generation, we cannot derive any classifying information. The top left figure illustrates the corresponding row vector in $\mathbf{w}_1$ for class 1 showing how certain elements (e.g. 24 and 34) are emphasized indicating their influence for classification. In particular element 34, which also has a relatively strong peak in the corresponding sources indicates this elements represents class. information for the 1 digit.

Similar illustrations are depicted for class 2 digits in the right figures. Here a few elements of the classification weights are emphasized and in particular element 26 also having a strong source peak again revealing characterizing classifying information.

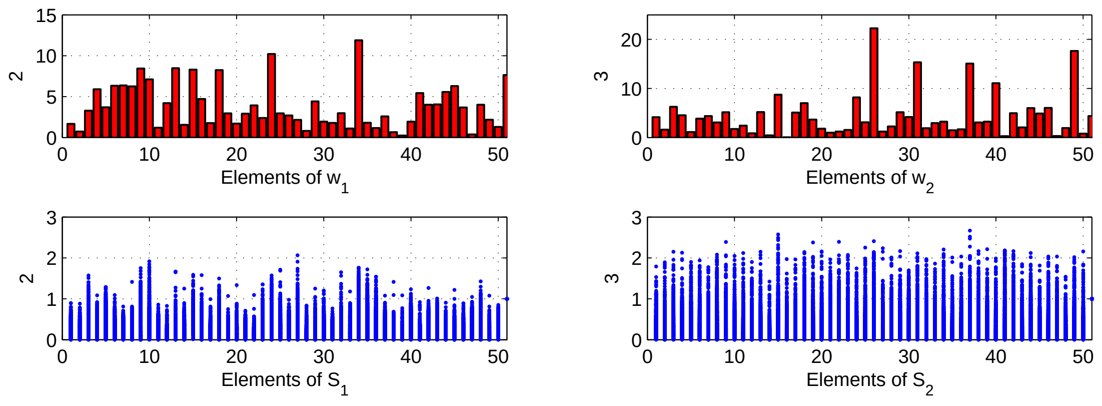

*Figure 5.13:* Weights in $\mathbf{W}$ and corresponding elements in $\mathbf{s}$ for digit 1 (*left*) and digit 2 (*right*) with parameters $\Theta = \{50, 0, 0, 0, 1\}$.

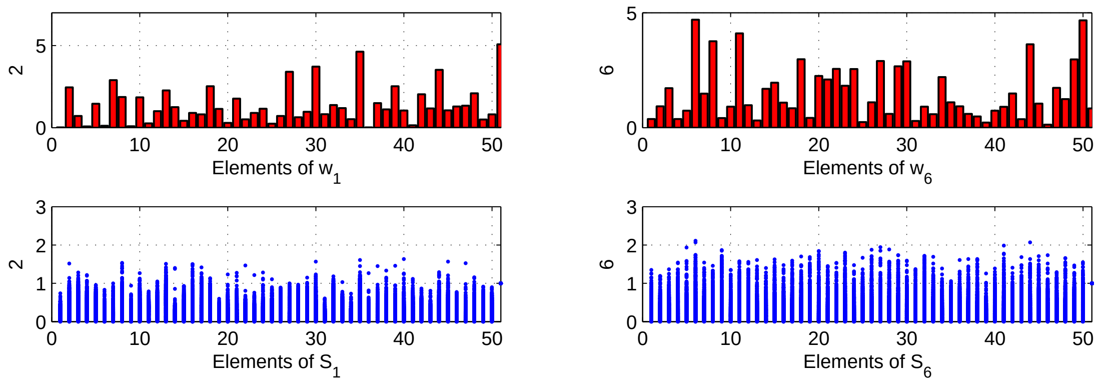

*Figure 5.14:* Weights in $\mathbf{W}$ and corresponding elements in $\mathbf{s}$ for digit 1 (*left*) and digit 6 (*right*) with parameters $\Theta = \{50, 0, 0, 0, 1e-3\}$.

Using the balance parameter $\sigma^2$ to regulate the classification of $\mathbf{s}$ a similar analysis can be made as depicted in figure 5.14 for digit 1 and 6. From these illustrations the same classifying elements can be found, but any clear evidence for the improved classification test error cannot be identified as a result of the regulation.

The classification weights can also be viewed as features by propagating the rows of $\mathbf{W}$ back through the generative network, i.e. $\mathbf{x}_w = \mathbf{A} \cdot \mathbf{W}^{\mathrm{T}}$. The corresponding weight features for $\sigma^2 = \{1, 1e-3\}$ shown in figure 5.15 can be seen as prototype features used for classification. In comparing the two regulations a small improvement can be seen for most digits and in particular for digit 8.

*Figure 5.15:* Row vectors of $\mathbf{W}$ propagated as features for $\sigma^2 = 1$ (*left*) and $\sigma^2 = 1e-3$ (*right*)

The regulation dependency for larger network is illustrated in figure 5.16. The middle illustration clearly shows how the optimal network size for this set of parameters $\Theta = \{d, 0, 0, 0, 1e-5\}$ has increased to around $d = 100$ compared to $\sigma^2 = 1$ as in figure 5.6.

We devise two immediate strategies for selecting network sizes in term of optimal class. performance. Choosing the network $d = 100$ leads to an initial optimal size expecting little or zero regulation of $\mathbf{A}$ or $\mathbf{W}$. Alternatively a large suboptimal network, e.g. $d = 500$, requirering stronger regulation may have the potential to outperform the smaller network and may encourage more sparse features. In the further analysis we will try to make an analysis of this problem by comparing both strategies. Hence for the small layer we choose $d = 100$ hidden units and $\sigma^2 = 1e-3$ and for the large layer we choose $d = 500$ hidden units and $\sigma^2 = 1e-5$.

## Regulation of classification weights ($\gamma$)

Regulation with $\sigma^2$ can only constrain the classification information in the sources $\mathbf{s}$ to a certain optimal level

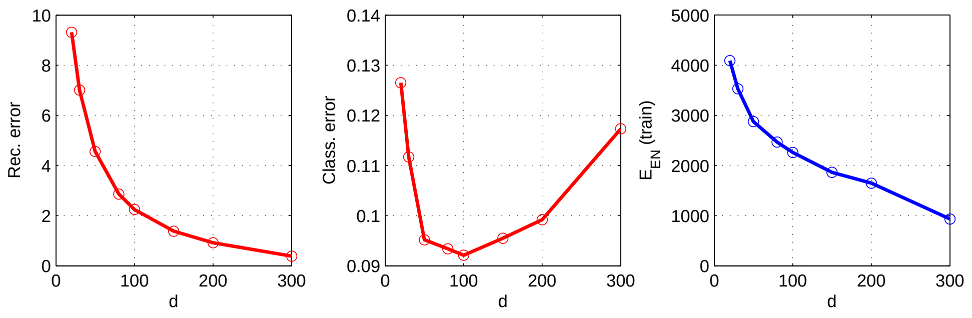

*Figure 5.16:* Performance for different sizes of hidden layers d with parameters $\Theta = \{d, 0, 0, 0, 1e-5\}$. *Left:* Reconstruction error (test). *Mid:* Class. error (test). *Right:* Class. error (train).

holding max. classification information. Further optimization can be achieved by regulating the classification weight matrix **W**. The performance of the network for different regulations is illustrated in figure 5.17 initially showing how the reconstruction error is unaffected as expected.

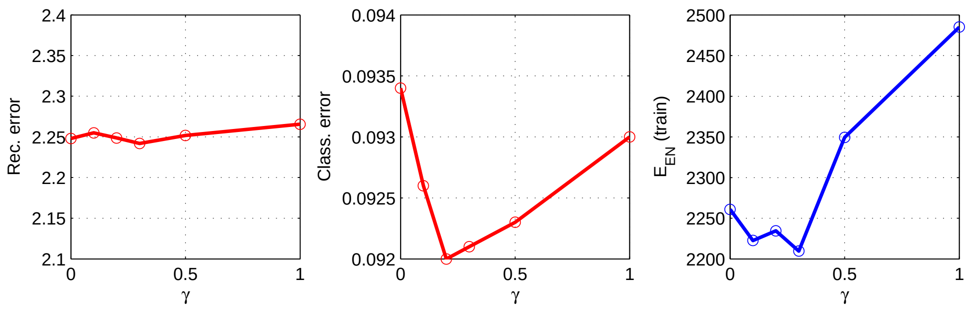

*Figure 5.17:* Performance for different regulations of the weight **W** with parameters $\Theta = \{100, 0, 0, \gamma, 1e-3\}$. *Left:* Reconstruction error (test). *Mid:* Class. error (test). *Right:* Class. error (train).

More importantly the middle figure reveals an overfit for small regulations and a bias for stronger regulations as the discriminants become more linear. This can also be seen from the right figure showing a smaller training error for light regulations. Conversely the higher trainingerror for stronger regulations fails to capture the underlying structure of the training data and thus induces a bias. An optimal regulation can be identified at $\gamma = 0.2$.

*Figure 5.18:* Classification weights for $\gamma = 0$ (*left*) and $\gamma = 0.2$ (*right*) with parameters $\Theta = \{100, 0, 0, \gamma, 1e-3\}$.

From figure 5.18 showing the corresponding weights **W** as features for zero and optimal regulation no clear difference can be seen and considering the very low classification improvement, this is also expected.

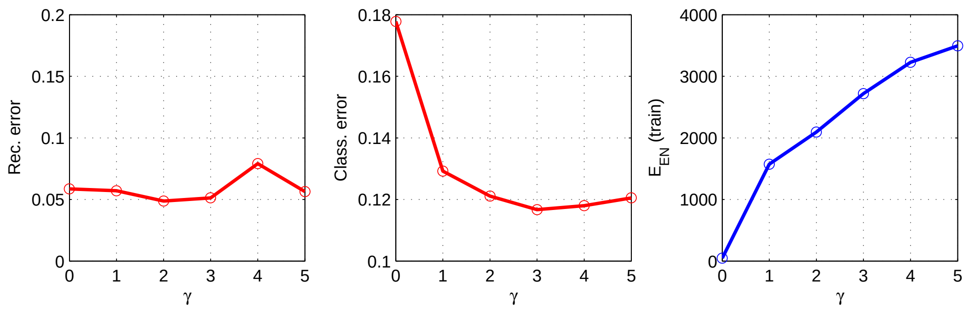

*Figure 5.19:* Performance for different regulations of the class. weights **W** with parameters $\Theta = \{500, 0, 0, \gamma, 1e-5\}$. *Left:* Reconstruction error (test). *Mid:* Class. error (test). *Right:* Class. error (train).

Similar effects can be seen for the larger network $d = 500$ in figure 5.19. The class. test error (middle figure) reveals an optimal regulation at $\gamma = 3$ showing how larger networks require stronger regulation as expected. Despite the regulation this larger network still has inferior class. performance compared to the smaller network $d = 100$.

*Figure 5.20:* Classification weights for $\gamma = 0$ (*left*) and $\gamma = 0.2$ (*right*) with parameters $\Theta = \{500, 0, 0, \gamma, 1e-5\}$.

From figure 5.18 showing the corresponding weights $\mathbf{W}$ as features for zero and optimal regulation only a slight difference can be spotted.

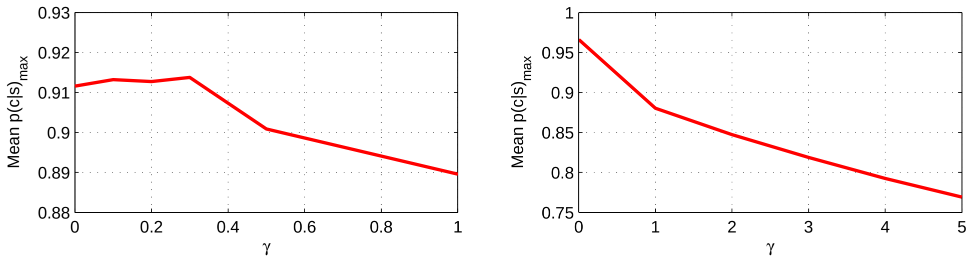

*Figure 5.21:* Mean of max $p(c|\mathbf{s})$ for all samples with parameters $\Theta = \{100, 0, 0, \gamma, 1e-5\}$ (*left*) and $\Theta = \{500, 0, 0, \gamma, 1e-5\}$ (*right*)

A different way to evaluate the effect of regulating the weights in $\mathbf{W}$ is to analyze how much the model classifies the different classes in terms of the softmax output probability $p(c|\mathbf{s})$. Figure 5.21 shows how the mean class. probability decreased as expected as the regulation increases leading to more linear discriminants.

## Regulation of generative weights ($\beta$)

As a final parameter of our model any potential overfit of the codebook features $\mathbf{A}$ can be controlled by the regulation parameter $\beta$. Regulation of $\mathbf{A}$ can in addition also provide more sparse features compared to those in figure 5.3.

To illustrate the effect of regulation each row in figure 5.22 show the extracted features for $d = 100$ latent variables and for different degrees of regulations of $\beta$.

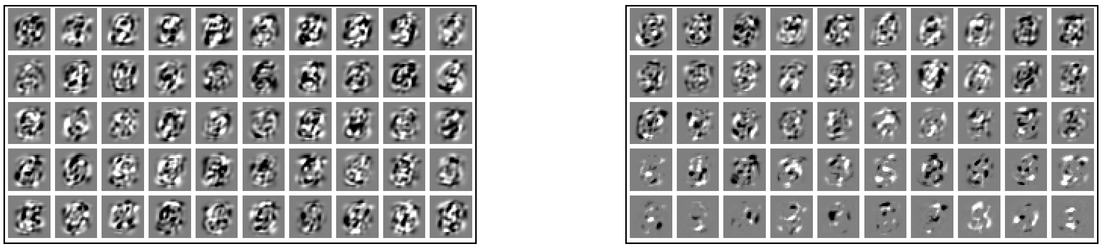

*Figure 5.22:* Extracted features with parameters $\Theta = \{100, 0, \beta, 0, 1e-3\}$ for regulation of $\beta = \{0, 0.2, 0.7, 1, 2, 5, 10, 20, 50, 100\}$ (starting top left row).

It is clear to see how the unconstrained features become more sparse for stronger degrees of regulation. For the strongest regulation shown with $\beta = 100$ the features does not quite resemble localized oriented filters as in the simple cells of V1. The exhibitory and inhibitory parts can still be identified, but are not localized. With a threshold of $10\%$ the mean SI is shown in the right figure of 5.23 for both $\mathbf{A}$ and $\mathbf{S}$ and in conjunction with the features shown above the mean SI reveals similarly more sparse features for stronger regulation through $\beta$ as expected.

The mean SI for $\mathbf{S}$ in the left figure shows how the sources become less sparse for stronger regulation. As the codebook features become more sparse and thus smaller in energy as shown top in figure 5.23 more elements in s are requires for proper representation of the observed data as seen at the bottom of the figure.

Eventhough we have achieved sparse unconstrained features with the regulation the final evaluation must be conducted on the testerror figures. The corresponding test-errors shown in figure 5.24 reveal how the regulation

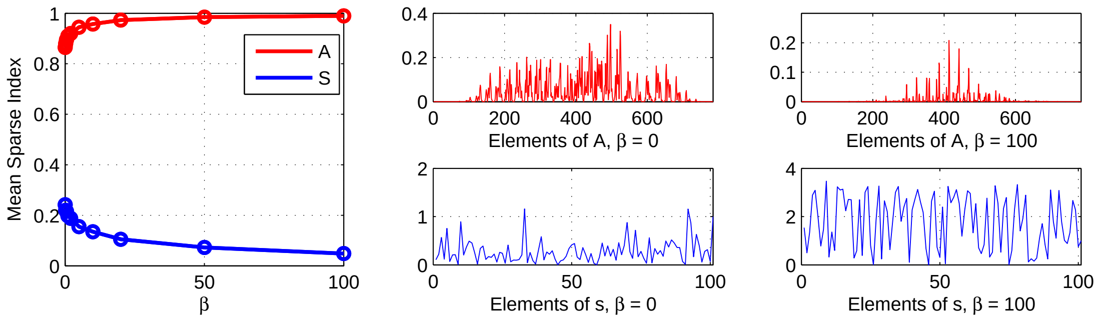

*Figure 5.23:* The mean sparsity index of **A** & **S** (*left*) and examples of corresponding features (*top*) and sources (*bottom*) with parameters $\Theta = \{100, 0, \beta, 0, 1e-3\}$ for no and strong regulation of $\beta$.

of the features has resulted in degraded classification performance. In addition the reconstruction error has also increased indicating that the sparse features does not give a better representation. As the sources become more sparse for stronger regulation of **A** we also induce a bias, which results in degraded representation in the sources leading to worse classification as can be seen right in figure 5.24.

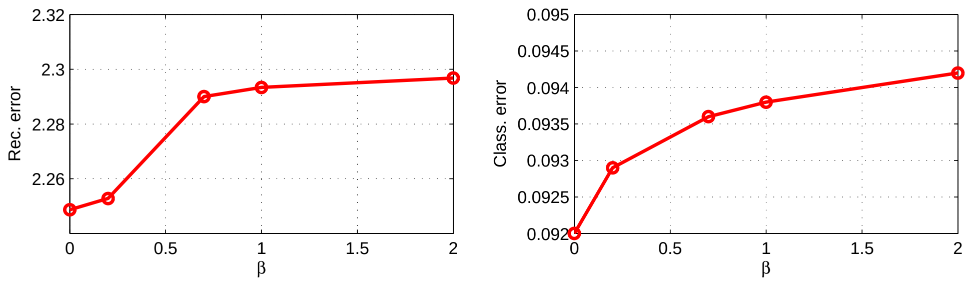

*Figure 5.24:* Test errors for different degrees of regulation of $\beta$ with parameters $\Theta = \{100, 0, \beta, 0, 1e-3\}$.

For the larger network of $d = 500$ hidden variables the corresponding codebook features illustrated left in figure 5.25 clear shows very noisy patches indicating too strong regulation. Damping the influence of the balance parameter to $\sigma^2 = 1e-3$ results in similar features as shown right in figure 5.25.

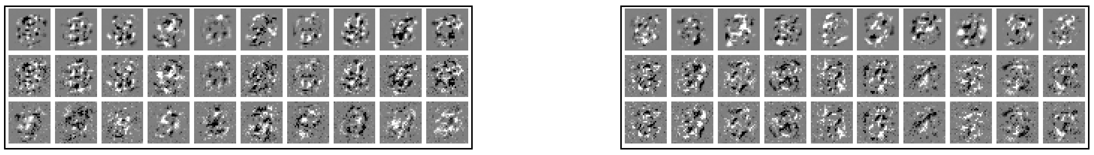

*Figure 5.25:* Extracted features for regulation of $\beta = \{10, 20, 30\}$ (starting top left row) with parameters $\Theta = \{500, 0, \beta, 0, 1e-5\}$ (*left*) and $\Theta = \{500, 0, \beta, 0, 1e-3\}$ (*right*).

From the cost function given in eq. (4.22) we see how the regulations by $\beta$ and $\sigma^2$ are inverse proportional and linear. Hence any regulation using $\sigma^2$ and $\beta$ during update can in principle be downscaled by adjusting an appropriate stepsize $\eta$. Thus the noisy results achieved in figure 5.25 could indicate a `MATLAB` instability, but further investigations are required to pinpoint the cause. Removing the regulation induced by the balance parameter to $\sigma^2 = 1$ we obtain features similar to those for $d = 100$ shown in figure 5.26.

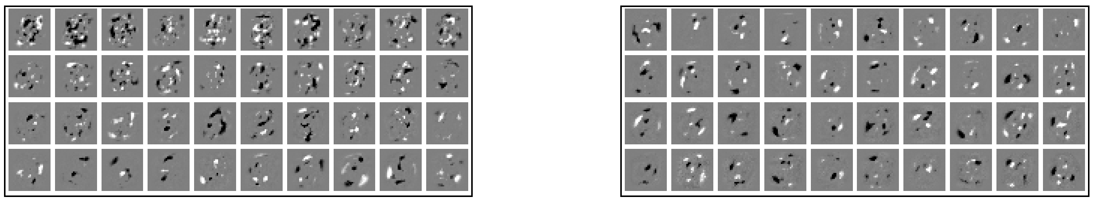

*Figure 5.26:* Extracted features with parameters $\Theta = \{500, 0, \beta, 0, 1\}$ for regulation of $\beta = \{10, 20, 50, 70, 100, 120, 150, 200\}$ (starting top left row).

It is clear to see how the unconstrained features become more sparse for stronger degrees of regulation. With a threshold of $1\%$ and $10\%$ the mean SI is shown in figure 5.27 for both $\mathbf{A}$ and $\mathbf{S}$ respectively and in conjunction with the features shown above the mean SI reveals similarly more sparse features for stronger regulation through $\beta$ as expected.

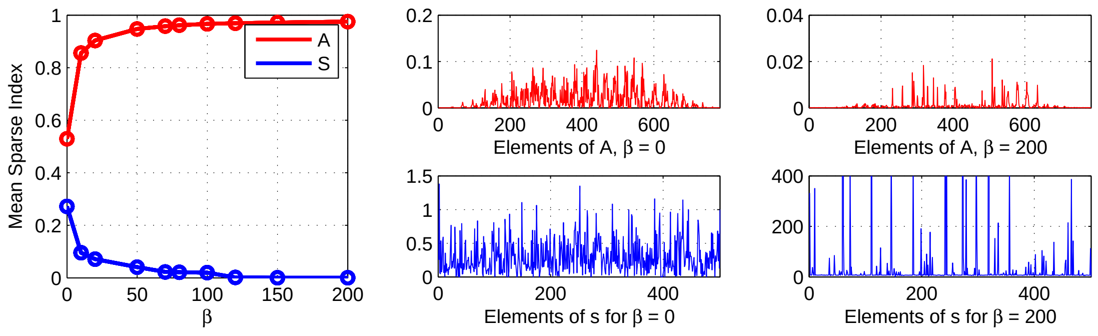

*Figure 5.27:* The mean sparsity index of $\mathbf{A}$ & $\mathbf{S}$ (*left*) and examples of corresponding features (*top*) and sources (*bottom*) with parameters $\Theta = \{500, 0, \beta, 0, 1\}$ for zero and strong regulation of $\beta$.

In addition the mean SI for the sources $\mathbf{S}$ in the left figure also become less sparse for stronger regulation as also seen for the smaller network in figure 5.23. As the energy in the features $\mathbf{A}$ decrease the sources compensate with larger elements and due to the non-linearity of $f(\mathrm{s})$ we get these extremely strong distinct elements.

The corresponding test-errors shown in figure 5.28 reveal an optimum at $\beta = 70$, where the class. testerror is lowest.

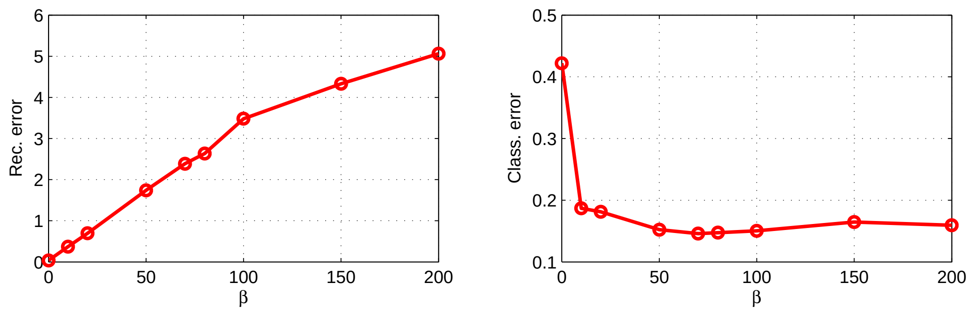

*Figure 5.28:* Test errors for different degrees of regulation of $\beta$ with parameters $\Theta = \{500, 0, \beta, 0, 1\}$.

The figure also reveals how the improved classification is achieved at the expense of higher reconstruction error, where stronger regulation leads to worse reconstruction as also seen for the smaller network in figure 5.24. Despite improved results achieved with $\sigma^2 = 1$, we conjecture that the larger network with $d = 500$ hidden units and sufficient regulation does not have superior classification performance compared to the smaller network $d = 100$.

Summary

For the initial layer we have seen how most of the different regulations optimized empirically improved the classification error rate. The analysis showed how the sources were biased wrt. to reconstruction, but overfitted wrt. classification. Regulation with the balance parameter $\sigma^2$ proved efficient to reduce the classification overfit without affecting the reconstruction error.

We have tested two different strategies for selecting network size and initial simulations indicated an optimal network size at appr. $d = 100$ in terms of classification test error requirering little or no regulation. In contrast large network suffering from overfit showed inferior class. performance despite strong regulation attempts.

Thus for the initial layer a network of $d = 100$ latent variables and parameters
$\Theta_1 = \{d = 100, \alpha = 0, \beta = 0, \gamma = 0.2, \sigma^2 = 1e-3\}$ is identified as optimal and the performance for this model is given in table below.

To complete the analysis layer 1 the non-linear mapped sources with these parameters are shown in figure 5.29 illustrates how the mapping function induces non-linearity on the sources.

| L1 Model | Rec. error | Class. error |
|---|---|---|
| $\{100, 0, 0, 0.2, 1e-3\}$ | 2.25 | 9.2% |

*Table 5.1:* L1 performance.

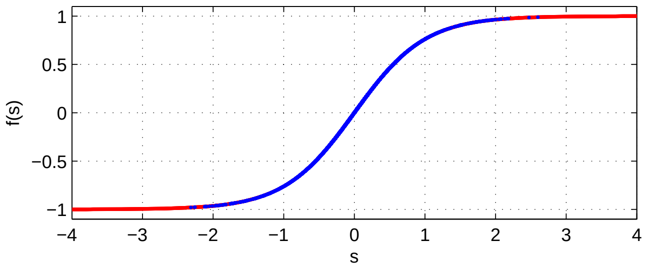

*Figure 5.29:* Mapping of sources onto $f(s)$ with parameters $\Theta = \{100, 0, 0, 0.2, 1e-3\}$.

In the case of very small sources we can equate the mapping function $f(\mathbf{s}) = \mathbf{s}$ and neglecting the softmax converting function, our classifier can be characterized as linear. Comparing with the official class. result of the MNIST dataset a linear classifier (1-layer NN) achieves $12\%$ error rate. Our small improvement indicates the positive effect of the non-linearity as shown in figure 5.29.

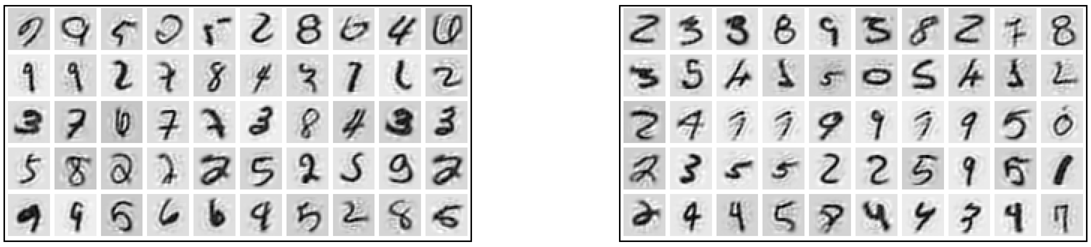

*Figure 5.30:* Subset of misclassified digits from model with parameters $\Theta = \{50, 0, 0, 0.2, 1e-3\}$. Each row represents each class starting from top left row representing digits misclassified of zero.

To evaluate the final performance figure 5.30 illustrates a subset of the misclassified digits for the optimal parameters of layer 1. It is evident to see how the misclassified digits are similar in structure as the corresponding class. For class 1 (2nd row left) all digits are slim and narrow as the 1 digit and class 9 (5th row right) include as few 4 digits, which bear resemble for a 9 digit.

### 5.2.2 Non-negative Constrained Model

Evaluating the classification error rates the unconstrained model show superior performance compared to the non-negative model (NN) as also evident from figure 5.1 showing how the unconstrained model achieve lower training-error for that particular model. This suggests that the non-negative constraint prohibits the model from learning both the generative features in $\mathbf{A}$ and in particular the classification weights in $\mathbf{W}$ sufficiently due to the multiplicative update rules in eq. (4.40) restricting the weights are $\mathbf{W}$ to non-negativity.

In figure 5.31 the performance of the NN model is shown for different size of latent variables. It clearly illustrate how the model is suffering from a bias for both reconstruction and classification and has worse performance in both measures compared to the UN model (figure 5.6).

A subset of features in $\mathbf{A}$ is shown in in each row in figure 5.32 with corresponding reconstruction for the different classes. Both figures depict features becoming more sparse yielding improved reconstruction for larger models as expected.

To better evaluate the sparsity of NN model figure 5.33 depicts the mean sparsity index with a threshold on $10\%$. The figure clearly illustrate how the features become more sparse in alignment with figure 5.32, but also shows how the sources are proportional in terms of sparsity. As the codebook features become more sparse and small more patches are required active in the sources to reconstruct the observed data as the non-negative constrain only allows additive reconstruction.

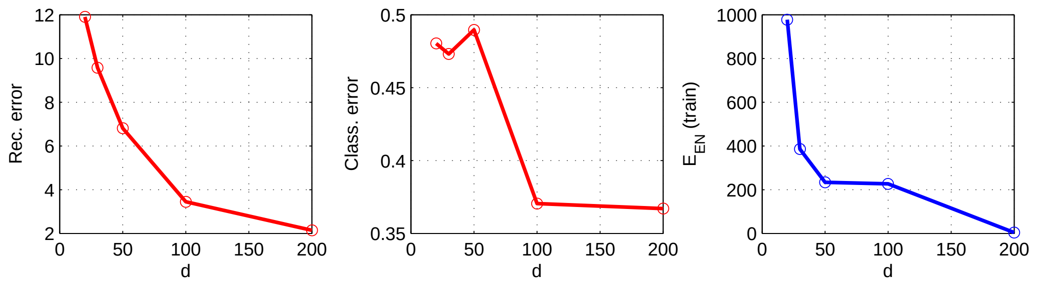

*Figure 5.31:* Test errors for different network sizes for the NN model with parameters $\Theta = \{d, 0, 0, 0, 1\}$.

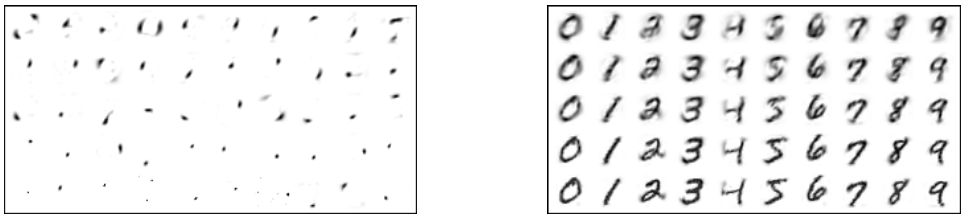

*Figure 5.32:* Features (*left*) and reconstruction of digits (*right*) for different NN network sizes $d = \{20, 30, 50, 100, 200\}$ (each row respectively) with parameters $\Theta = \{d, 0, 0, 0, 1\}$.

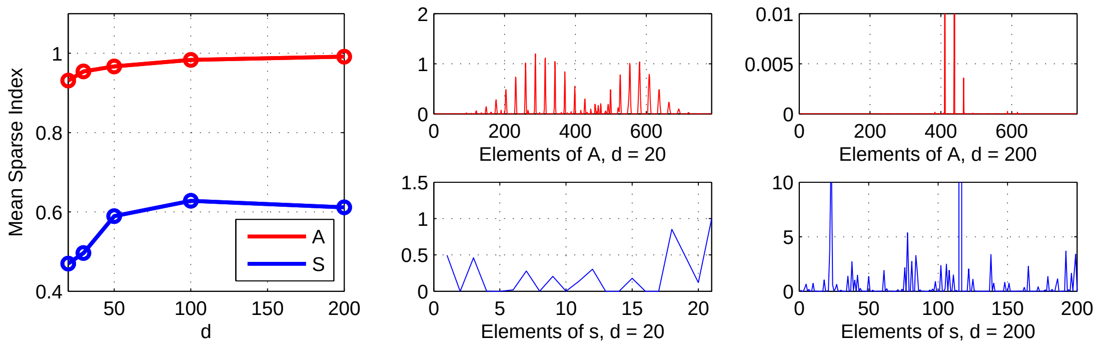

*Figure 5.33:* The mean sparsity index of $\mathbf{A}$ & $\mathbf{S}$ (*left*) and examples of corresponding features (*top*) and sources (*bottom*) with parameters $\Theta = \{d, 0, \beta, 0, 1\}$ for different layer sizes d.

To accommodate for the non-negative constraint on the weights we conduct a dual training session, where the network features $\mathbf{A}$ and sources $\mathbf{S}$ are learned strictly unsupervised without the classifier. Afterwards the classification weights $\mathbf{W}$ are trained both non-negative and unconstrained keeping the sources constant.

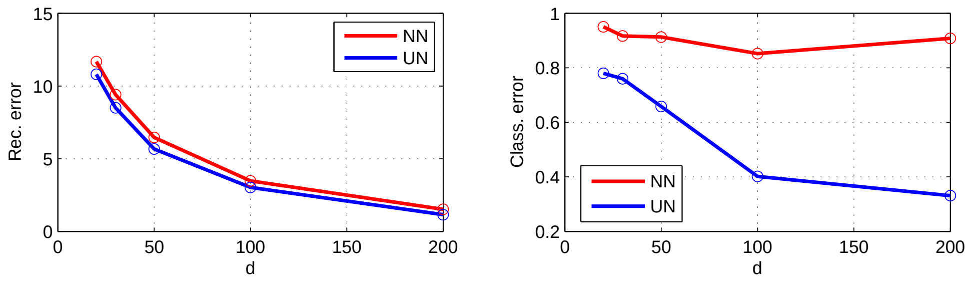

*Figure 5.34:* Dual session training. *Left*: Reconstruction error for different network sizes. *Right*: Classification errors for both NN and UN model. All with parameters $\Theta = \{d, 0, 0, 0, 1\}$.

The performance shown in figure 5.34 clearly illustrates how the class. and reconstruction performance is degraded for the NN model compared to figure 5.31. This highlights the significance of including classification information in the sources to achieve good classification performance, as mentioned. In addition the unconstrained weights $\mathbf{W}$ also give improved classification compared to the non-negative constrained, as expected.

### 5.2.3 Generation of digits

As efficient classification is based on proper generative representation in the sources s optimizing model parameters for lowest class. error as opposed to reconstruction energy seems justified. Thus for a model with good class. performance we expect equivalently good generative properties.

For the small network $d = 100$ with optimal parameters, where $\alpha = 0$ figure 5.35 shows generated digits $0 \dots 9$ each initialized randomly for different stopping criteria defined as a threshold of the class association $p(c|\mathbf{s})$, denoted $y_{stop}$.

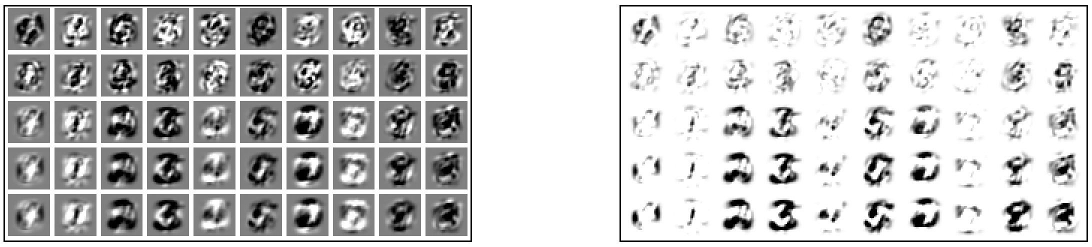

*Figure 5.35:* Generation of digits $0 \dots 9$ (each column) for stop criterias $y_{stop} = \{0.90, 0.98, 0.99, 1e-4, 1\}$ (each row) and parameters $\Theta = \{100, 0, 0, 0.2, 1e-3\}$. *Left*: Full dynamic range. *Right*: Negative pixels truncated.

For the small stop criterias $y_{stop}$ we generate very noisy digits hard to recognize and for increasing $y_{stop}$ we can see how the quality improves. It is also evident to see how digits 2, 3, 5 and 8 are generated with sufficient quality to be recognized. In addition the digits are very similar and seemed based on the same prototype. For $y_{stop} = 1$ (bottom row) the generator never reach this stopping criteria and thus conducts a full convergence only stopped by max. iterations of 10000. With $y_{stop} = 1$ we achieve the best quality of digits and in figure 5.36 5 sessions with $y_{stop} = 1$ is shown.

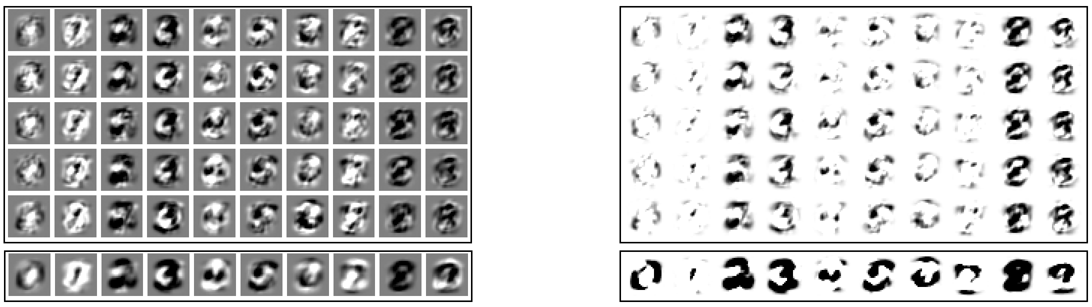

*Figure 5.36:* Generation of digits $0 \dots 9$ (each column) for $y_{stop} = 1$ and different sessions (each row). Bottom row shows corresponding weights $\mathbf{W}$ as features. All with parameters $\Theta = \{100, 0, 0, 0.2, 1e-3\}$. *Left*: Full dynamic range. *Right*: Negative pixels truncated.

From the figure it is clear to see how the digits all appear the same, i.e. we achieve the same minimum with very little variance. In addition this model is without any regulation of the sources $\alpha = 0$, i.e. no prior information is given to potentially generate more varying digits. Comparing with the prototype features from $\mathbf{W}$ (bottom row) we see how the generated digits highly resemble these features. As our generation is solely based on classification information from exactly these weights (eq. (4.51)), this connection is expected. To compare with a network with lower classification error rate figure 5.37 depicts generated digits for the suboptimal network $d = 100$ without any regulation.

The generated digits are clearly not of same quality as figure 5.36 and cannot be compared with the class. weight features (bottom rows). This suggests that best class. performance also give best generation of digits and vice versa as the model then has sufficient representation of the data.

## 5.3 Deep Networks

The whole concept of using a deep network is to learn increasingly more abstract features from layer to layer. It is therefore essential that the relevant information is preserved up through the layers, e.g. classification or

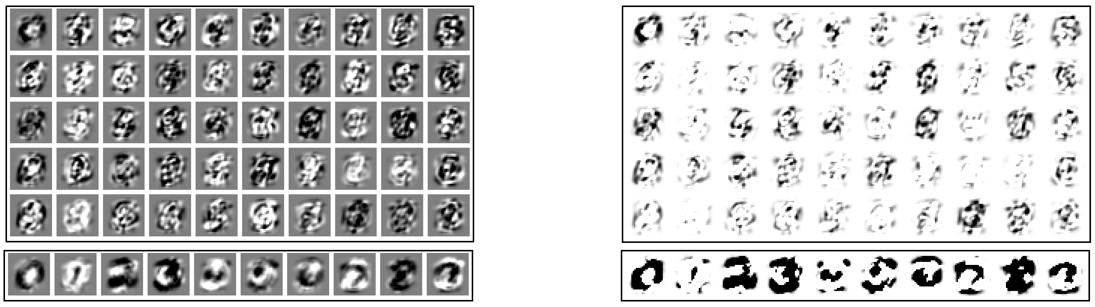

*Figure 5.37:* Generation of digits $0 \dots 9$ (each column) for $y_{stop} = 1$ and different sessions (each row). Bottom row shows corresponding weights $\mathbf{W}$ as features. All with parameters $\Theta = \{100, 0, 0, 0, 1\}$. *Left*: Full dynamic range. *Right*: Negative pixels truncated.

reconstruction information. If a layer is too narrow or small we risk neglecting important information, which cannot be recovered in subsequent layers. In contrast if the layer is too wide we might obtain a degree of redundancy leading to a potential overfit learning the same representation or identity of the previous layer. This is particularly relevant if not regulation is enforced in subsequent layers.

In contrast a wide layer may provide an overcomplete sparse representation of the data, which can lead to several potential advantages as argued by Ranzato et. al. [25]. A sparse representation can provide a simple interpretation of the input by a small number of distinct parts of the cognitive extracted features. Using high dimensional overcomplete features can in addition lead to increased likelihood of separating data categories or classification.

From the initial layer with $d_1 = 100$ latent variables we build a shallow deep net by adding an extra layer including the classifier such that the network structure becomes 784-100-$d_2$-10, where the final layer is the classifying network. The impact of the size of layer 2 is illustrated in figure 5.38 showing the class. overfit for larger networks evident from the right figure.

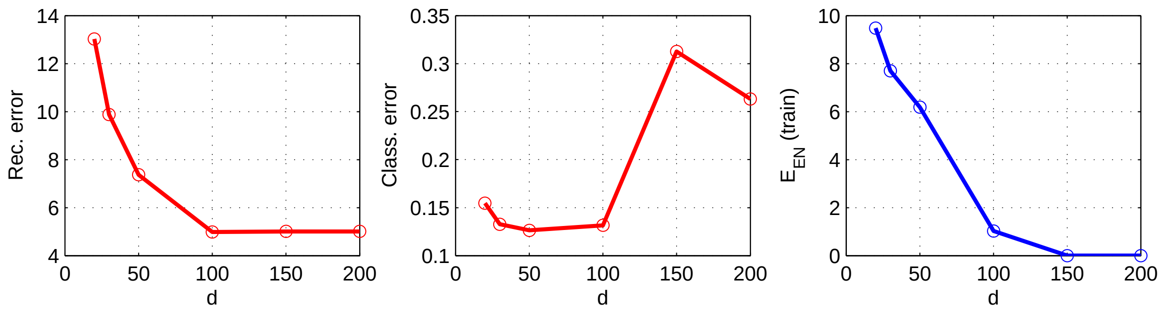

*Figure 5.38:* Performance for different layer sizes $d$ with parameters $\Theta_2 = \{d, 0, 0, 0, 1\}$. *Left:* Reconstruction error (test). *Mid:* Class. error (test). *Right:* Class. error (train).

In the left illustration the reconstruction test error is based on propagating the error through the network down to the observed $\mathbf{x}$, i.e. $e_r = \frac{1}{2N} \parallel \mathbf{X} - \mathbf{A}_1\mathbf{A}_2\mathbf{S}_2 \parallel^2$. For layers over the critical size of $d = 100$ the figure reveals how the reconstruction test error falls to a constant indicating an optimal representation of the sources from layer 1 with no regulation. In addition we see that the class. test error is fact worse than the initial layer with $e_c = 9.2\%$.

From the figure we can identify the small network with $d = 50$ latent variables as the optimal size and conducting a similar analysis as for layer 1 for the different parameters of regulation $\theta = \{\alpha, \beta, \sigma^2, \gamma\}$ we achieve the classification performance shown in figure 5.39.

Despite the hard attempt the figures revel how the class. performance has indeed improved settling at $e_c = 10.4\%$ (reconstruction error $e_r = 7.8\%$), but not sufficiently to outperform layer 1. Thus adding this additional layer has not given the class. improvement as hoped for.

The right illustration in figure 5.40 depicts the propagated codebook features in $\mathbf{A}_2$ for a network size of $d = 50$ and the left figure shows the corresponding codebook features $\mathbf{A}_1$ for layer 1 for comparison. The features for

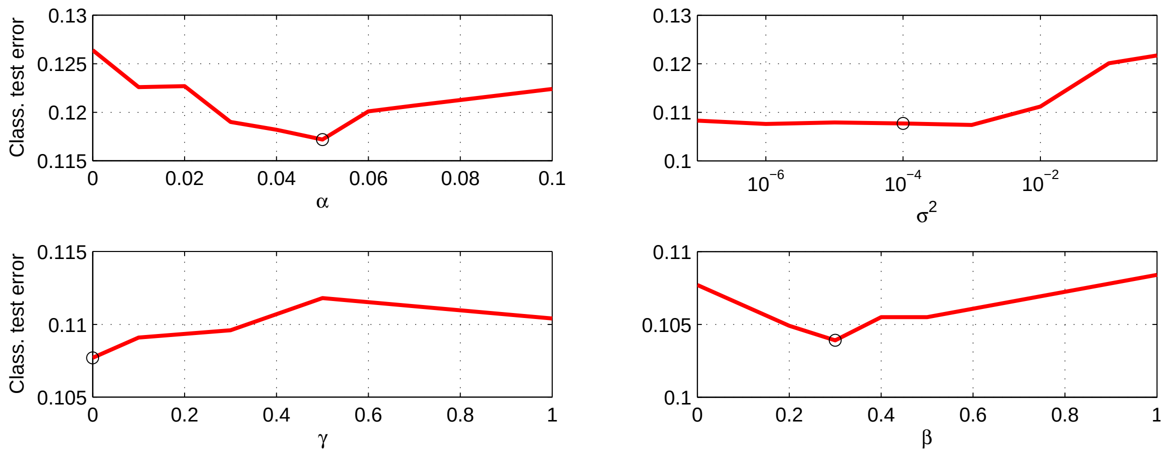

*Figure 5.39:* Classification test errors for layer 2 for $d = 50$ hidden units.

layer $\mathbf{A}_2$ are still characterized as very noisy and complex, but comparing with $\mathbf{A}_1$ a slightly more meaningful texture can be spotted with smaller spatial frequency and some even with digit resemblance.

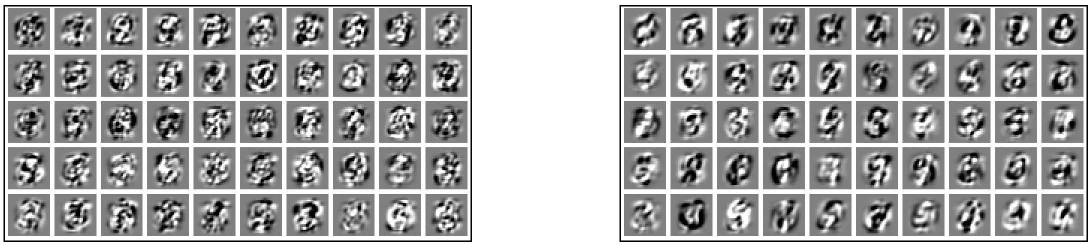

*Figure 5.40:* Codebook features for layer 1 with parameters $\Theta_1 = \{100, 0, 0, 0.2, 1e-3\}$. (*left*) and layer 2 with parameters $\Theta_2 = \{50, 0.05, 0.3, 0, 1e-3\}$.

To further analyze this network figure 5.41 illustrate the weights for class 0 revealing few correlations, e.g. elements 1 and 14, as seen before in layer 1.

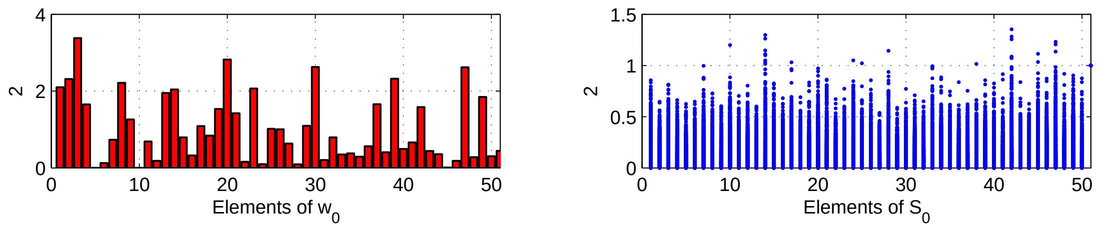

*Figure 5.41:* Weights in $\mathbf{W}$ (*left*) and corresponding elements in $\mathbf{s}$ (*right*) for digit 0 with parameters $\Theta_2 = \{50, 0.05, 0.3, 0, 1e-3\}$.

A more interesting approach is to analyze the weights as features as shown in figure 5.42. The patches has clearly improved compared to layer 1 and can be recognized as digits in alignment with the less noisy codebook features in figure 5.40. Despite the quality of these weight features the class. performance has not improved as already seen.

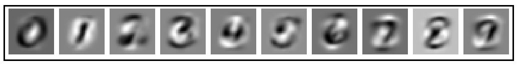

*Figure 5.42:* Classification weight as features for layer 2 with parameters $\Theta_2 = \{50, 0.05, 0.3, 0, 1e-3\}$.

As a different strategy we analyze the much larger network with $d = 200$ hidden variables. Figure 5.43 shows the class. error for the different regulations. Initially it can be seen how stronger regulation is required due to the larger network size of $d = 200$. Unfortunately the class. error of $e_c = 11.0\%$ has not improved compared to both L1 and the smaller L2 network with $d = 50$ units.

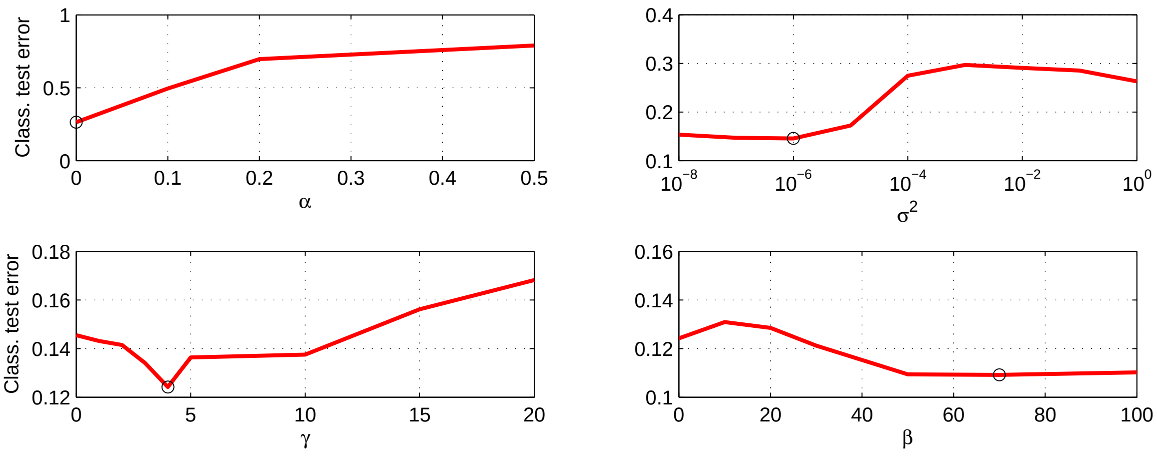

*Figure 5.43:* Classification test errors for layer 2 for $d = 200$ hidden units.

In addition the regulation of the sources in the top left illustration is optimal for $\alpha = 0$. This is in opposition with the light regulation required for the smaller network in figure 5.39 suggesting the empirical regulation probing being too coarse for $d = 50$.

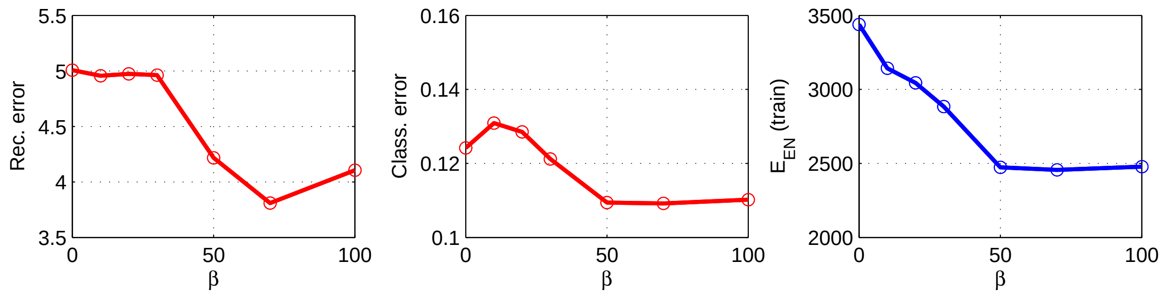

*Figure 5.44:* Test errors for different degrees of regulation of $\beta$ with parameters $\Theta_2 = \{200, 0.0, 70, 4, 1e-6\}$.

More interestingly is the evolution of the reconstruction test error for different regulations of the features in $\mathbf{A}$ as shown in figure 5.44.

As other regulations using $\gamma$ and $\sigma^2$ had no significant impact on the rec. test error (not shown), the main contribution of the rec. bias identified in figure 5.38 for larger layers is caused by overfit of the codebook features $\mathbf{A}$ and sources $\mathbf{S}$. This is due to the width of layer 2 being larger than L1 and thereby allowing an overcomplete representation of the data and thus a larger risk of overfit as already discussed.

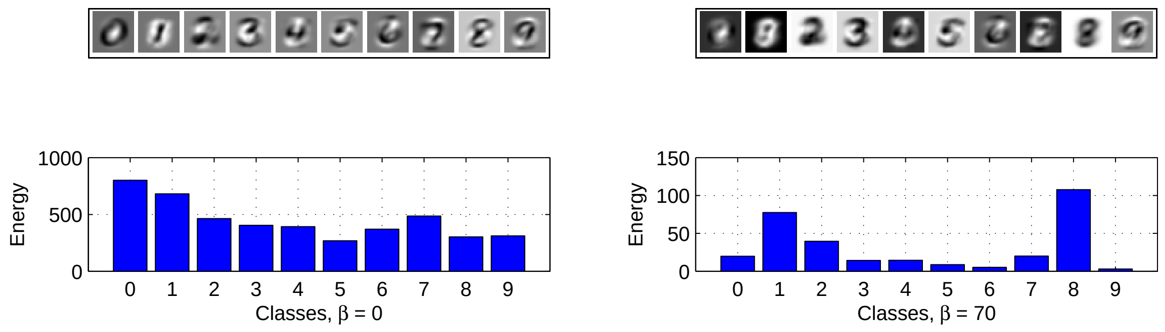

*Figure 5.45:* Class. weights as features for $\beta = 0$ (*top left*) and $\beta = 70$ (*top right*) and corresponding energy (bottom) with parameters $\Theta_2 = \{200, 0, \beta, 4, 1e-6\}$.

This can also be viewed from the corresponding class. weight features for the different regulations of $\mathbf{A}$ as depicted in the top of figure 5.45. It is clear to see how the weight features seem fairly generalized for all classes, but become more tuned for specific classes for stronger regulations, i.e. classes 2,3 and 5. This is also evident from the corresponding energy distribution showing how classes 2,3 and 8 are strongly represented.

| Class | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $\beta = 0$ | 42 | 73 | 63 | 91 | 123 | 87 | 94 | 168 | 367 | 134 | 1242 |
| $\beta = 70$ | 60 | 54 | 98 | 107 | 137 | 146 | 89 | 104 | 184 | 113 | 1092 |

*Table 5.2:* Misclassified digits for each class for network with parameters $\Theta_2 = \{200, 0, \beta, 4, 1e-6\}$.

This indicates an overfit for these classes and is also seen from the amount of misclassified digit for these classes listed in table 5.2 showing a relatively high error rate. Conversely the classes 0 and 1 with poorly recognizable weight features in figure 5.45 has relatively small error rate.

In our experiments we have in all cases had the classifier network attached, i.e. all parameters were trained in a single session. Additional simulations without the classifier attached we trained the network in a dual session, i.e. initially $\mathbf{A}$ and $\mathbf{S}$ and afterwards the weights $\mathbf{W}$. For both the smaller $d = 50$ and larger network $d = 200$ we achieve higher class. test error of $e_c = 19.3\%$ and $e_c = 18.95\%$ respectively. This indicates that to obtain good class. performance the sources must be optimized supervised with class labels during training for each layer as also seen for the non-negative constrained model.

In the experiments conducted by Bengio et. al. in [2] it was shown for a classification task of the MNIST dataset (refer to section 5.1) that the classification structure was only necessary in the initial layer. Once the classification information was learned by the network it meant to stay and hence subsequent layers was only optimized wrt. reconstruction [2].

To summarize we have analyzed two different strategies for choosing the size of additional layer of our deep network model. As for layer 1 the large size of $d = 200$ hidden units of layer 2 with sufficient regulation did not achieve a class. error rate superior to the smaller size of $d = 50$ hidden units. Thus for the second layer the small network with parameters $\{d = 50, \alpha = 0.05, \beta = 0, \gamma = 0.3, \sigma^2 = 1e-4\}$ is identified as optimal for classification (still performing horribly) and the performance for both layer sizes can be seen in table 5.3.

| L2 Model | Rec. error | Class. error |
|---|---|---|
| $\{50, 0.05, 0, 0.3, 1e-4\}$ | 7.8% | 10.4% |
| $\{200, 0, 70, 4, 1e-6\}$ | 3.8% | 11.0% |

*Table 5.3:* L2 performance.

The quality and class dependent structure of the misclassified digits for both layer sizes (not shown) are of similar structure as for layer 1 illustrated in figure 5.30.

### 5.3.1 Generation of digits

With the second layer added we generate a set of digits for both $d = 50$ and $d = 200$ hidden units. Setting the stop criteria $y_{stop} < 1$ we experienced similar results as for layer 1 (not shown), where we generated noisy digits as seen in figure 5.35. Hence we only conduct our analysis with $y_{stop} = 1$.

For the small network $d = 50$ and large $d = 200$ hidden units figures 5.46 and 5.47 depict a set of generated digits each initialized randomly. The quality of the digits are as seen before very similar to the weight features $\mathbf{W}$ as expected.

Thus considering the influence of the visual quality of the weight features and that the larger network $d = 200$ with no regulation of the codebook features $\beta = 0$ has far better generalized weight features, figure 5.45 shows a subset of generated digits of far better quality. Still we generate basically the same digit for different initializations as seen before.

To summarize adding the second layer we learned more structured codebook features allowing a small improvement on the generative properties of the model. This had a direct impact on the quality of the generated digits compared to layer 1, but caused the class. error rate to increase.

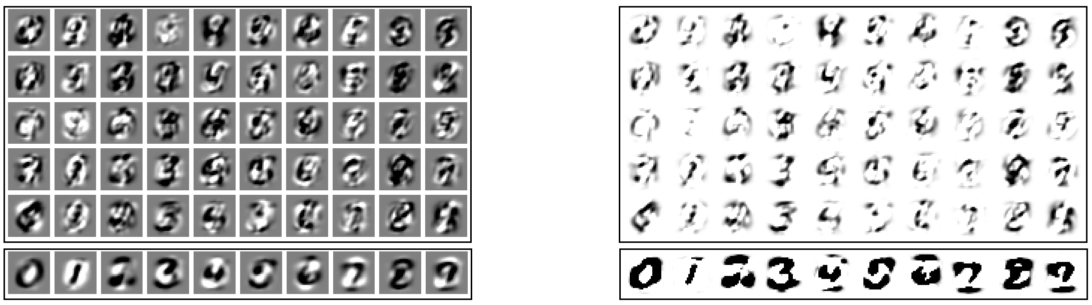

*Figure 5.46:* Generation of digits $0 \dots 9$ (each column) for $y_{stop} = 1$ and different sessions (each row). Bottom row shows corresponding weights $\mathbf{W}$ as features. All with parameters $\Theta_2 = \{50, 0.05, 0.3, 0, 1e-4\}$. *Left*: Full dynamic range. *Right*: Negative pixels truncated.

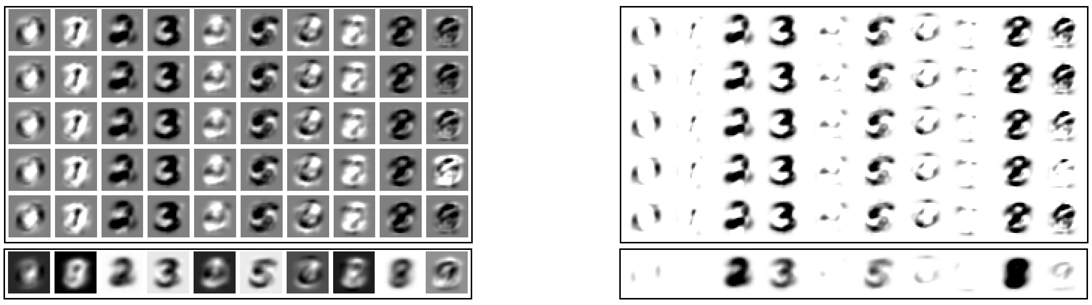

*Figure 5.47:* Generation of digits $0 \dots 9$ (each column) for $y_{stop} = 1$ and different sessions (each row). Bottom row shows corresponding weights $\mathbf{W}$ as features. All with parameters $\Theta = \{200, 0, 70, 4, 1e-6\}$. *Left*: Full dynamic range. *Right*: Negative pixels truncated.

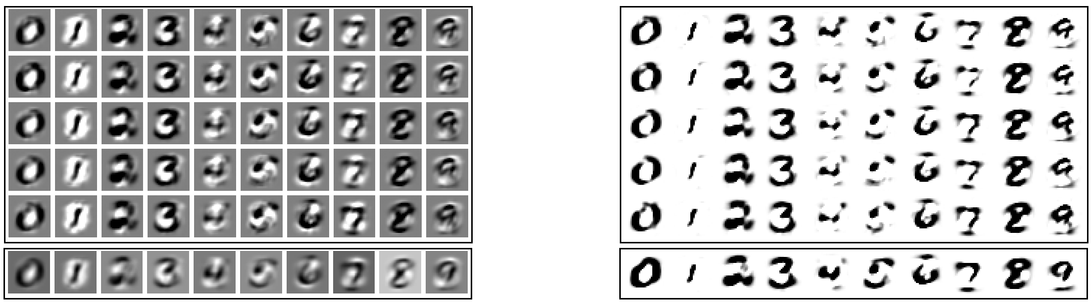

*Figure 5.48:* Generation of digits $0 \dots 9$ (each column) for $y_{stop} = 1$ and different sessions (each row). Bottom row shows corresponding weights $\mathbf{W}$ as features. All with parameters $\Theta = \{200, 0, 0, 4, 1e-6\}$. *Left*: Full dynamic range. *Right*: Negative pixels truncated.

From our analysis we have seen how our deep network model greatly suffers from generating digits of sufficient quality. The few digits generated were basically small variants of the same prototype digits stored as classification weights in $\mathbf{W}$ and thus our model does not have the ability to generate different digits.

The prior of the sources $p(\mathbf{s})$ is highly complex and modelling it with a simple Laplace distribution has proven to be highly insufficient. Digits of high quality has been generated successfully in the deep belief net by Hinton et al. [12] by modelling this complex toplevel prior with a infinite undirected net.

## 5.4 Summary

In the extensive analysis of our deep network we have shown the overall performance and limitations in terms of its generative and discriminative properties, inspite of a relatively limited number of simulations and a shallow net.

With classification performance as our criteria we have build a 2-layer deep net model based on two different strategies. From the analysis smaller networks with little regulation showed best class. performance compared to larger networks with stronger regulation for both layers.

Cognitive features extracted from NMF proved limited in both classification and reconstruction. The unconstrained features of L1 were very abstract structured and could not be characterized as cognitive, but proved superior for both class. and rec. error rates. compared to the non-negative features due to their exhibitory and inhibitory effects. In analyzing the initial layer we achieved a classification error rate $e_c = 9.2\%$ and comparing with the MNIST hall of fame this is slightly better than a linear classifier (1-layer NN) of $12\%$.

By adding a second layer we achieved less abstract features and in testing the same strategy for determining an optimal size of L2 the smaller network also proved best in terms of class. error rate, whereas the larger layer size showed better reconstruction due to the overcomplete representation as $d_{L2} > d_{L1}$. The resulting classification performance of the second layer did not improve the overall classification compared to the initial layer. The misclassified digits for L2 (not shown) were of same character as for L1, where the misclassified digits showed similar structure as the true class data.

One of the major simplifications in our deep network is the modelling of the top level priors $p(\mathbf{s})$ using the simple Laplace distribution. In the deep belief net of Hinton et al. this top-level prior is modelled by a two layer RBM equating an infinite net with tied weights allowing for complex modelling and generation of digits. This simplification of ours have a direct impact on the generation of digits as the latent sources are extracted from this top-level prior.

The optimal model from L1 were optimized with no regulation on the sources $\alpha = 0$ and thus models the source priors with a uniform distribution. This meant the generated digits were only based on classification information and thus the resulting digits resembled the prototype class. weights with little variance. However the less abstract codebook features in L2 lead to class. weights resembling recognizable digits and hence digits generated from L2 were more recognizable than L1, but still of poor quality. The choice using the Laplace distribution for simplified modelling of the source priors has indeed proven inferior.

To achieve efficient classification performance throughout the layers of the deep network model, the latent sources must be trained supervised with class labels in each layer. It seems class. information is not maintained sufficiently between the layers for our model in direct opposition with results found by Bengio et al. [2], where only training the initial layer supervised was necessary.

Comparing with the deep belief net by Hinton et al. the subsequent retraining revise lower level weights that were learned first to fit in better with the weights that were learned later. This is achieved relatively easy as the deep belief net allows simple propagation up and down the layers implemented by complementary priors. Due to the structure of our deep network model such upward propagation is not simple and thus we have omitted the fine-tuning of model parameters. However in evaluating the performance of our model such retraining could seem necessary to achieve better class. error rate and in particular for improving the second layer.

Throughout our analysis we have also seen the importance of learning good generative representation of the data in order to achieve good classification. Our deep network model can in general not be characterized to have any high classification performance due to the simplifications made compared to the deep nets based on Hinton et al. [12].

# 6. Conclusion

Mathematical modelling of the visual cognition system and in particular the primary visual cortex has received much attention throughout the years. In this context this thesis introduced the area of modelling cognitive representations for visual data.

We presented cognitive components as unsupervised grouping of data such that the ensuing group-structure is well-aligned with that resulting from human cognitive activity [8]. Using generative models has the appealing property of allowing a strong representation of the underlying structure of observed data. In this respect we build different generative models in our cognitive analysis based on cognitive components extracted unsupervised from NMF, MF both linear and non-linear.

An efficient cognitive generative model will in this sense capture the underlying structure of the cognitive components to generate new data. We proposed two different model types as candidates for such efficient cognitive models, namely *The Mixture Models* and *Deep Network Models*.

Mixture Models

The principle of building advanced models from simpler ones is an easy and very efficient approach in modelling complex structures. Based on a linear model we extracted sparse cognitive features using NMF. The first mixture model grouped the sources using K-Means and proved highly inefficient as it only captured high level correlation between the features and lead to very poor generation of new digits.

As a natural step the more advanced Gaussian mixture model (MoG) was introduced as a potential improvement. Maximum likelihood learning of the model was described in terms of the *Expectation-Maximization* algorithm as tractable approach for training mixture model parameters. The MoG model generated far better digits, but failed to model the feature correlation efficiently.

Extending the MoG model to include dimension reduction of the data by introducing the factor analyzer as a new kernel function resulted in a slight improvement of the generated digits, as it better captured subdimensional structures.

In general the class of mixture model described in this thesis all suffered from lack of complexity in capturing the underlying structure of the visual data and hence proved inefficient for modelling visual cognitive data.

Deep Network Models

Deep belief nets introduced by Hinton et al. has proven superior as a generative model for cognitive data also used for classification [12]. This model induced complementary priors as an essential concept allowing for very flexible propagation through the network layers.

The concept of using a generative model for classification tasks is in general an efficient approach as it allows a stronger representation of the full structure of observed data. In this respect we have given a short description of the principles of Hinton's deep belief model to serve as a motivation for proposing a simpler deep network model limited to only generative propagation.

A theoretical framework was presented as individual modules with different modelling properties used to build a deep network model. By training each layer separately unsupervised we have analyzed a shallow structured deep net in terms of classification and generation of digits.

Non-negative sparse features characterized as cognitive components proved limited in performance due to the absence of inhibitory parts. Our model was therefore based on unconstrained abstract features not characterized as cognitive.

The generative properties of the deep network model greatly suffered from the simplification of modelling the source priors with a simple Laplace distribution and became particularly evident from our experiments. Hence efficient modelling of top-level priors is essential in deep network models to achieve good generative properties. Using a generative representation for classification has the possibility of projecting data onto subdimensional manifolds allowing for greater separation and thus simpler discrimination. In our experiments we achieved poor classification performance slightly better than a 1-layer linear classifier for the MNIST dataset.

As our model suffered from simple recognition propagation we omitted any subsequent fine-tuning of model parameters. For a shallow deep network we experienced degraded performance for additional layers suggesting the essential need for a subsequent interlayer fine-tuning. We conjecture the absence of the 2nd stage training has been a major contributor to the overall degraded performance of our deep network model.

## 6.1 Outlook

The concept of deep networks has proven very powerful as both a generative and discriminative model. In terms of modelling the processing of the visual cortex these model also resemble the same overall structure of gradually learning more abstract representations. Hence future variants of our model should point in the direction of deep belief network as introduced by Hinton et al. [12].

With the motivation of modelling features seen in the visual cortex, i.e. Gabor-like filters, an additional module can based on the constraint of independent sources. Using ICA for visual data has shown to produce these sparse localized structures resembling receptive fields in V1 [1] [14].

Our implementation of the learning of the deep network model is based on the simple 1st order gradient descent approach. This method may guarantee a local optimum provided sufficiently small stepsizes, but are often very slow due to the gradient becoming smaller as it approaches a solution. Training the model based on 2nd order optimization, such as Newton-based methods, may reduce simulation times and serves as a straightforward improvement.

Initially our model is build from greedy learning without any subsequent interlayer dependent fine-tuning. Omitting such post-training certainly has a an impact on overall performance as the model parameters will not be optimized all together, but only individually. Hence incl. a retrofit of model parameter for fine-tuning may serve as a major improvement.

# References

[1] M. Arngren. Analyzing sensory neurons by extracting features from color image using linear models. Technical report, Technical University of Denmark, 2006.

[2] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle. Greedy layer-wise training of deep networks. *Technical Report 1282, Département d'Informatique et Recherche Opérationnelle, Université de Montréal*, 2006.

[3] Jeff. A. Bilmes. A gentle tutorial of the em algorithm and its application to parameter estimation for gaussian mixture and hidden markov models. *ICSI-TR-97-021*, April 1997.

[4] Christopher M. Bishop. *Neural Network for Pattern Recongition*. Oxford Univerity Press, 1995.

[5] Matteo Carandini, Jonathan B. Demb, Valerio Mante, David J. Tolhurst, Yang Dan, Bruno A. Olshausen, Jack L. Gallant, and Nicole C. Rust. Do we know what the early visual system does? *The Journal of Neuroscience*, 25(46):10577–10597, Nov. 2005.

[6] L. Feng and L. K. Hansen. Phonemes as short time cognitive components. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP'06)*, volume V, pages 869–872, may 2006.

[7] Zoubin Ghahramani and Geoffrey E. Hinton. The em algorithm for mixures of factor analyzers. *Technical Report CRG-TR-96-1, University of Toronto*, 1997.

[8] L. K. Hansen, P. Ahrendt, and J. Larsen. Towards cognitive component analysis. In Finnish Cognitive Linguistics Society Pattern Recognition Society of Finland, Finnish Artificial Intelligence Society, editor, *AKRR'05 -International and Interdisciplinary Conference on Adaptive Knowledge Representation and Reasoning*. Pattern Recognition Society of Finland, Finnish Artificial Intelligence Society, Finnish Cognitive Linguistics Society, jun 2005. Best paper award AKRR'05 in the category of Cognitive Models.

[9] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Verlag, 2001.

[10] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science Magazine*, Vol 18, No. 7, July 2006.

[11] Geoffrey E. Hinton. To recognize shapes, first learn to generate images. *Technical Report UTML TR 2006-004*, Oct, 2006.

[12] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, Vol. 313. no. 5786, July 2006.

[13] Patrik O. Hoyer. Modelling receptive fields with non-negative sparse coding. *Computational Neuroscience: Trends in Research*, 52-54:547–552, 2003.

[14] Patrik O. Hoyer and Aapo Hyvärinen. Independent component analysis applied to feature extraction from colour and stereo images. *Network: Computation in Neural Systems*, 11, August, 2000.

[15] D. H. Hubel and T. N.Wiesel. Receptive fields and functional architecture of monkey striate cortex. *Journal of Physiology*, 195:215–243, 1968.

[16] Aapo Hyvärinen, Juha Karhunen, and Erkki Oja. *Independent Component Analysis*. John Wiley & Sons Inc., 2001.

[17] Daniel D. Lee and H. Sebastian Seung. Learning the parts of objects by nonnegative matrix factorization. *Nature*, 40:788–791, 1999.

[18] Daniel D. Lee and H. Sebastian Seung. Algorithms for non-negative matrix factorization. *Neural Information Processing Systems (NIPS)*, pages 556–562, 2000.

[19] David J. C. MacKay. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.

[20] K. Madsen, H.B. Nielsen, and O. Tingleff. *Optimization with Constraints, 2nd Edition*. DTU/IMM, March 2004.

[21] R. E. Madsen, L. K. Hansen, and O. Winther. Singular value decomposition and principal component analysis. Technical report, 2004.

[22] Morten Mørup and Mikkel N. Schmidt. Sparse non-negative matrix factor (2-d) deconvolution. May 2006.

[23] Radford M. Neal and Geoffrey E. Hinton. A view of the em algorithm that justifies incremental, sparse and other variants. 1998.

[24] Pedersen Petersen. The matrix cookbook. *Technical paper*, 2007.

[25] M. Ranzato, C. Poultney, S. Chopra, and Y. LeCun. Efficient learning of sparse representations with an energy-basedmodel. *Advances in Neural Information Processing Systems*, (19), 2006.

[26] Mikkel N. Schmidt and Morten Mørup. Sparse non-negative matrix factor 2-d deconvolution for automatic transcription of polyphonic music. Aug. 2006.

[27] Gideon Schwarz. Estimating the dimension of a model. *The Annals of Statistics*, 6(2):461–464, Mar. 1978.

[28] Eero P. Simoncelli and Bruno A. Olshausen. Natural image statistics and neural representation. *Annual Review of Neuroscience*, 24:1193–216, May 2001.

[29] M. E. Tipping and C. M. Bishop. Probabilistic principal component analysis. *Journal of the Royal Statistical Soceity*, Part 3(Series B 61):611–622, 1999.

# A. Appendix

This appendix is a collection of small mathematical derivations useful for the main analysis' in the report, some self-contained.

## A.1 Appendix - Principal Component Analysis

One of the most simple decompositions is the *Principal Component Analysis* (PCA) with two important properties used in information processing, namely *whitening* and *dimension reduction*, where whitening is defined as the process of removing any mean-value from the data, ensuring unity variance and decorrelating the data (i.e. zero covariance). This technique of whitening is used widely as a pre-processing step in data analysis as it allows 1st and 2nd order information to be reduced or aligned for datasets from different sources.

The basic concept in PCA is to transform a random variable $\mathbf{x}$ onto a set of orthogonal axis (denoted the principal components) based on maximizing 2nd order information or variance of $\mathbf{x}$. In this context we will describe PCA both as a pre-processing step in terms of whitening data and as an efficient tool for dimension reduction of data.

Let us denote the dataset $\mathbf{X} = \{\mathbf{x}_n\}_{n=1}^{N}$, where $\mathbf{x}_n$ is an $m$ dimensional column vector, i.e. $\mathbf{X}$ is an $m \times N$ matrix. In this context, we introduce the *expectation* operator $\mathcal{E}(\cdot)$ as statistical averaging given by

$$\mathcal{E}\{\mathbf{x}\} = \int_{-\infty}^{\infty} \mathbf{x}\, p_{\mathbf{x}}(\mathbf{x})\, \mathrm{d}\mathbf{x} \tag{A.1}$$

where $p_{\mathbf{x}}(\mathbf{x})$ denoted the probability density function for $\mathbf{x}$.

The sample mean $\mu_i$ can easily be determined as the 1st order moment by $\mu_i = \mathcal{E}\{\mathbf{y}_i\}_{i=1}^{m}$, where $\mathbf{y}_i$ in this case is the $N$-dimensional i'th row vectors of $\mathbf{X}$. As the data $\mathbf{x}_n$ is usually based on measurements, the probability density $p_{\mathbf{x}}(\mathbf{x})$ may not always be known. The mean vector $\mu = \{\mu_1, \mu_2, \ldots, \mu_m\}$ can in these cases be approximated empirically by

$$\mu \approx \widehat{\mu} = \frac{1}{N}\sum_{n=1}^{N} \mathbf{x}_n \tag{A.2}$$

Similarly the covariance is determined by the 2nd order central moment (2nd order moment with zero mean) by $\mathbf{C}(x_k, x_l) = \mathcal{E}\{(x_k - \mu_k)(x_l - \mu_l)\}$, for $k, l = 1, 2, \ldots, m$. For $k = l$, the covariance becomes $\mathbf{C}(x_k, x_k) = \sigma_k^2$, i.e. the variance of $x_k$. The corresponding *covariance matrix* $\mathbf{\Sigma}$ for $m = 2$ can be expressed as

$$\mathbf{\Sigma} = \left[\begin{array}{cc} \sigma_1^2 & \mathbf{C}(x_1, x_2) \\ \mathbf{C}(x_2, x_1) & \sigma_2^2 \end{array}\right] \tag{A.3}$$

From the definition of $\mathbf{C}$ it is easy to see $\mathbf{C}(x_1, x_2) = \mathbf{C}(x_2, x_1)$ and that $\mathbf{\Sigma}$ is symmetric and always positive semi-definite. This can similarly be approximated empirically unbiased by

$$\widehat{\mathbf{\Sigma}} \approx \frac{1}{N-m}\sum_{n=1}^{N}(\mathbf{x}_n - \widehat{\mu})(\mathbf{x}_n - \widehat{\mu})^T \tag{A.4}$$

With the definition of the covariance matrix, it is easy to see that a white dataset has the identity matrix $\mathcal{I}$ as the covariance matrix, i.e. $\mathbf{\Sigma} = \mathcal{I}$, where the data components have unity variance and zero covariance.

To decorrelate the data $\mathbf{X}$, we decompose the covariance matrix into $\mathbf{\Sigma} = \mathbf{U}\mathbf{\Lambda}\mathbf{U}^{\mathbf{T}}$, which can be re-written to :

$$\mathbf{\Sigma u}_i = \lambda_i \mathbf{u}_i \tag{A.5}$$

where $\mathbf{U}$ holds the orthonormal eigenvectors $\mathbf{U} = \{\mathbf{u}_i\}_{i=1}^m$, where $\mathbf{UU}^T = \mathcal{I}$ and diag$(\mathbf{\Lambda}) = \{\lambda_i\}_{i=1}^m$ is the corresponding set of eigenvalues. Since $\mathbf{\Sigma}$ is always positive semi-definite, the eigenvalues are always non-negative. Geometrically this can be illustrated as in figure 3.1.

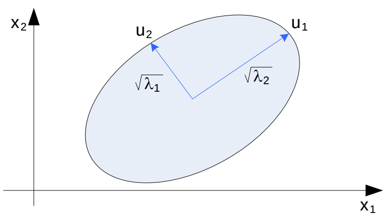

*Figure A.1:* Geometric illustration of the eigenvalue decomposition of the data subset $\mathbf{X}$ for dimensions $m = 2$. Here the ellipse depicts the constant joint probability density $p(\mathbf{x_1}, \mathbf{x_2})$ and the eigenvectors $u_1$ and $u_2$ denotes the directions of maximum variance with lengths proportional to the eigenvalues, $\lambda_1$ and $\lambda_2$.

In the eigenvalue decomposition, the orthonormal eigenvectors $\mathbf{U}$ denotes the directions of maximum variance (called the *Principal Components*) and the corresponding eigenvalues are proportional to their lengths, refer to [4].

Decorrelating the data corresponds to a coordinate transformation of the datapoints $x_{i,n}$ using the eigenvectors $\mathbf{U}$ with lengths $\mathbf{\Lambda}^{-\frac{1}{2}}$ as basis vectors, i.e. :

$$\mathbf{z}_i = \mathbf{\Lambda}^{-\frac{1}{2}}\mathbf{U}^T(\mathbf{x}_i - \mu) = \mathbf{V}(\mathbf{x}_i - \mu) \tag{A.6}$$

By sorting the eigenvectors wrt. eigenvalues in descending order, the dimension of the dataset $\mathbf{X}$ can easily be reduced to $d$ dimensions by projecting the datapoints $x_{i,n}$ onto a subset of the eigenvectors with the $d$ largest eigenvalues $\mathbf{U}'_d = \{\mathbf{u}_i\}_{i=1}^d$, where $d < M$.

This eigenvalue decomposition can also be achieved by *Singular Value Decomposition* (SVD), where the data itself is decomposed into $\mathbf{X} = \mathbf{U\Gamma V}^T$. Here $\mathbf{U}$ (symmetric $m \times m$ matrix) and $\mathbf{V}$ (symmetric $N \times N$ matrix) holds the ordered orthonormal eigenvectors for the column and row vectors in $\mathbf{X}$ respectively and the diagonal elements of $\mathbf{\Gamma}$ ($m \times N$ matrix) holds the corresponding shared singular values, refer to [21] for details. From the SVD we can construct a covariance matrix $\mathcal{E}\{\mathbf{XX}^T\} = \mathbf{U\Gamma V}^T\mathbf{V\Gamma U}^T = \mathbf{U\Gamma^2 U}^T$, which is exactly the eigenvalue decomposition with $\mathbf{\Gamma^2} = \mathbf{\Lambda}$. This means the decorrelation can be achieved by :

$$\mathbf{z}_i = \mathbf{\Gamma}^{-1}\mathbf{U}^T(\mathbf{x}_i - \mu) \tag{A.7}$$

where $\mathbf{\Gamma}$ and $\mathbf{U}$ can be reduced in size $d < m$ to obtain dimension reduction of the data.

In data analysis with high dimensional data $m > 2$, PCA can be a useful tool for reducing dimensions to $d = 2$ for visualization of the data. In such case the data $\mathbf{X}$ is projected into the 2 eigenvectors $\mathbf{U}_{2D} = \{\mathbf{u}_{max,1}, \mathbf{u}_{max,2}\}$ with the two largest eigenvalues $\{\lambda_{max,1}, \lambda_{max,2}\}$.

In the example below we illustrate the effect of *whitening* and *dimension reduction* using PCA.

**Example** :
*In this example, we have applied PCA to a small set of image data (*$1000$ *samples of size* $28 \times 28 \times 1$*), which will be described in details in section 3.1. Figure A.2 shows the scatterplots of the image data over dimension 1 and 2 and the corresponding 1st and 2nd principal axes after decorrelation.*

*The left illustration of figure A.2 clearly shows how the image data is strongly correlated since information of one variable gives information of the other. The dynamic range for the image data can also be seen as the scatterplot is bound from approximately ∼[0; 0.16]. The right figure show how the data is uncorrelated after whitening with unity variance. Information of one variable now gives no information of the other.*

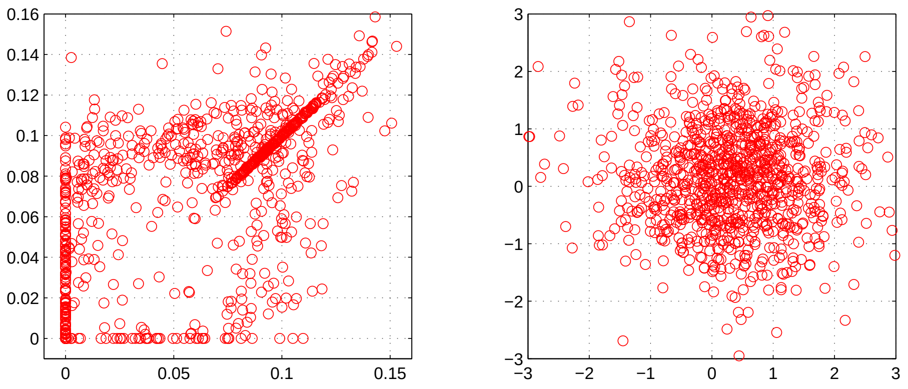

*Figure A.2: Left*: Scatterplot of un-processed image data for 1st and 2nd dimension. *Right*: Scatterplot of 1st and 2nd principal axes after whitening.

As an alternative approach, PCA can also be derived from a probabilistic approach, denoted *Probabilistic PCA* (PPCA), based on the same linear model as Factor Analysis (FA) with similar properties, presented later in section 2.5.1. PPCA will not be discussed here, but will be compared to FA shortly later and can be referred from [29]. The *Principal Component Analysis* presented will also later be applied to image data for easy visualization, as we shall see. In case of viewing covariance matrices, we show in appendix A.4 how these are projected into a set of principal components.

## A.2 Appendix - Bound in the EM algorithm

The "EM"-algorithm is based on estimating and increasing a bound $\mathcal{F}(p_{\mathbf{h}}, \mathbf{\Theta})$. In this section we will make a derivation of that bound.

By introducing the latent variable $\mathbf{h}$ denoting the unknown class labels, the log-likelihood for the data as defined in (2.15) can be redefined as the incomplete log-likelihood to

$$\mathcal{L}_{inc}(\mathbf{\Theta}) = \sum_n \ln p(\mathbf{x}_n|\mathbf{\Theta}) = \sum_n \ln \int p(\mathbf{x}_n|\mathbf{H}, \mathbf{\Theta})\ \mathrm{d}\mathbf{H} \tag{A.8}$$

where $\mathbf{H} = \{\mathbf{h}_n\}_{n=1}^N$ is the latent random variable invoked. By introducing the latent variables $\mathbf{h}_n$, we can simplify the maximization problem of the likelihood $\mathcal{L}_{inc}(\mathbf{\Theta})$. If we denote the distribution for the latent variables $p_{\mathbf{h}}(\mathbf{H})$ and assume a distinct distribution $p_{\mathbf{h}}(\mathbf{h}_n)$ for each sample $\mathbf{x}_n$, we can marginalize over the latent variables $\mathbf{h}_n$ and the incomplete log-likelihood $\mathcal{L}_{inc}(\mathbf{\Theta})$ becomes

$$\begin{aligned}
\mathcal{L}_{inc}(\mathbf{\Theta}) &= \sum_n \ln p(\mathbf{x}_n|\mathbf{\Theta}) && \text{(A.9)}\\
&= \sum_n \ln p(\mathbf{x}_n|\mathbf{\Theta}) \int p_{\mathbf{h}}(\mathbf{h}_n)\ \mathrm{d}\mathbf{h}_n && \text{(A.10)}\\
&= \sum_n \int p_{\mathbf{h}}(\mathbf{h}_n) \ln p(\mathbf{x}_n|\mathbf{\Theta})\ \mathrm{d}\mathbf{h}_n && \text{(A.11)}
\end{aligned}$$

since $\int p_{\mathbf{h}}(\mathbf{h}_n)\ \mathrm{d}\mathbf{h}_n = 1$ the introduction of $p_{\mathbf{h}}(\mathbf{h}_n)$ does not affect the log-likelihood $\mathcal{L}_{inc}(\mathbf{\Theta})$. We can further expand the probability $p(\mathbf{x}_n|\mathbf{\Theta}) = \frac{p(\mathbf{x}_n|\mathbf{\Theta})p(\mathbf{h}_n|\mathbf{x}_n,\mathbf{\Theta})}{p(\mathbf{h}_n|\mathbf{x}_n,\mathbf{\Theta})} = \frac{p(\mathbf{x}_n,\mathbf{h}_n|\mathbf{\Theta})}{p(\mathbf{h}_n|\mathbf{x}_n,\mathbf{\Theta})}$ and rewrite (A.11) into

$$\begin{aligned}
\mathcal{L}_{inc}(\mathbf{\Theta}) &= \sum_n \int p_{\mathbf{h}}(\mathbf{h}_n) \ln \left( \frac{p(\mathbf{x}_n, \mathbf{h}_n|\mathbf{\Theta})}{p(\mathbf{h}_n|\mathbf{x}_n, \mathbf{\Theta})} \frac{p_{\mathbf{h}}(\mathbf{h}_n)}{p_{\mathbf{h}}(\mathbf{h}_n)} \right) \mathrm{d}\mathbf{h}_n && \text{(A.12)}\\
&= \sum_n \int p_{\mathbf{h}}(\mathbf{h}_n) \ln \frac{p(\mathbf{x}_n, \mathbf{h}_n|\mathbf{\Theta})}{p_{\mathbf{h}}(\mathbf{h}_n)}\ \mathrm{d}\mathbf{h}_n + \sum_n \int p_{\mathbf{h}}(\mathbf{h}_n) \ln \frac{p_{\mathbf{h}}(\mathbf{h}_n)}{p(\mathbf{h}_n|\mathbf{x}_n, \mathbf{\Theta})}\ \mathrm{d}\mathbf{h}_n && \text{(A.13)}
\end{aligned}$$

The term inside the last summation is identified as the *Kullback-Leibler* divergence $\mathrm{KL}\big(p_{\mathbf{h}}(\mathbf{h}_n) \mid\mid p(\mathbf{h}_n|\mathbf{x}_n, \boldsymbol{\Theta})\big)$ and is characterized by always being non-negative. Refer to appendix A.3 for a more detailed description of the Kullback-Leibler divergence. This allows us to form a lower bound on the log-likelihood as

$$\mathcal{L}_{inc}(\boldsymbol{\Theta}) = \sum_n \int p_{\mathbf{h}}(\mathbf{h}_n) \ln \frac{p(\mathbf{x}_n, \mathbf{h}_n|\boldsymbol{\Theta})}{p_{\mathbf{h}}(\mathbf{h}_n)} \, \mathrm{d}\mathbf{h}_n + \sum_n \mathrm{KL}\big(p_{\mathbf{h}}(\mathbf{h}_n) \mid\mid p(\mathbf{h}_n|\mathbf{x}_n, \boldsymbol{\Theta})\big) \tag{A.14}$$

$$\geq \sum_n \int p_{\mathbf{h}}(\mathbf{h}_n) \ln \frac{p(\mathbf{x}_n, \mathbf{h}_n|\boldsymbol{\Theta})}{p_{\mathbf{h}}(\mathbf{h}_n)} \, \mathrm{d}\mathbf{h}_n \tag{A.15}$$

$$\triangleq \mathcal{F}(p_{\mathbf{h}}, \boldsymbol{\Theta}) \tag{A.16}$$

This important result shows that the introduction of the latent variables results in a lower bound (known as the *free energy* in statistical physics) on the incomplete likelihood, $\mathcal{F}(p_{\mathbf{h}}, \boldsymbol{\Theta}) \leq \mathcal{L}_{inc}(\boldsymbol{\Theta})$. This inequality can also be derived from *Jensen's Inequality* (this is the more popular approach in the literature, e.g. [4]), which is based on the concavity of the logarithm function, refer to [4] for a more detailed description. The bound $\mathcal{F}(p_{\mathbf{h}}, \boldsymbol{\Theta})$ can be further expanded into

$$\mathcal{F}(p_{\mathbf{h}}, \boldsymbol{\Theta}) = \sum_n \int p_{\mathbf{h}}(\mathbf{h}_n) \ln \frac{p(\mathbf{x}_n, \mathbf{h}_n|\boldsymbol{\Theta})}{p_{\mathbf{h}}(\mathbf{h}_n)} \, \mathrm{d}\mathbf{h}_n \tag{A.17}$$

$$= \sum_n \int p_{\mathbf{h}}(\mathbf{h}_n) \ln p(\mathbf{x}_n, \mathbf{h}_n|\boldsymbol{\Theta}) \, \mathrm{d}\mathbf{h}_n - \sum_n \int p_{\mathbf{h}}(\mathbf{h}_n) \ln p_{\mathbf{h}}(\mathbf{h}_n) \, \mathrm{d}\mathbf{h}_n \tag{A.18}$$

where the last term inside the sum can be identified as the differential entropy of $\mathbf{h}$, denoted $\mathcal{H}(\mathbf{h})$ (refer to appendix A.3). If we define the *complete* log-likelihood by $\mathcal{L}_c(\boldsymbol{\Theta}) = \ln p(\mathbf{X}, \mathbf{H}|\boldsymbol{\Theta})$, we can rewrite (A.18)

$$\mathcal{F}(p_{\mathbf{h}}, \boldsymbol{\Theta}) = \mathcal{E}_{\mathbf{h}}\{\mathcal{L}_c(\boldsymbol{\Theta})\} - \sum_n \mathcal{H}(\mathbf{h}) \tag{A.19}$$

where the first term is the expected value of the complete-data log-likelihood with respect to the latent variables $\mathbf{h}$. This bound function is the basis of the iterating "EM"-algorithm.

## A.3 Appendix - Kullback-Leibler divergence

For a random variable $x$ with probability distribution $p(x)$, the differential *entropy* $\mathcal{H}$ defined from information theory, expresses a degree of information that the observed variable $x$ gives [16] and is defined as

$$\mathcal{H}(x) = -\int p_x(\eta) \;\log p_x(\eta) \, \mathrm{d}\eta \tag{A.20}$$

The entropy has the essential property of becoming larger the more "random" or unpredictable the data $x$ is. This is one of the main reasons why the entropy is a fundamental building block in information theory, incl. the KL-divergence.

For two different distributions $p(x)$ and $q(x)$ the mutual information is an expression of how much information members from $p(x)$ has on members from $q(x)$. One way to measure the mutual information is the *Kullback-Liebler*-divergence (KL), which is based on the entropy and defines a quantity to compare the difference between two probability distributions, $p(x)$ and $q(x)$.

$$\mathrm{KL}(p||q) = -\int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} \, \mathrm{d}x \tag{A.21}$$

As KL is based on the entropy the KL-divergence is a non-negative function and is only zero if and only if the two distributions $p(x)$ and $q(x)$ are equal. In addition the KL-divergence is asymmetric, i.e. $\mathrm{KL}(p||q) \neq \mathrm{KL}(q||p)$.

## A.4 Appendix - Visualizing multivariate covariance matrices

In this appendix section we show how a multivariate covariance matrix $\mathbf{\Sigma}$ can be visualized in a 2D environment via *Principal Component Analysis* (PCA), refer to section A.1 for details.

Given an $M$ dimensional random variable $\mathbf{x} = \{x_m\}_{m=1}^{M}$ with mean $\mu_x$ and covariance $\mathbf{\Sigma}_x$, we decompose the covariance matrix $\mathbf{\Sigma}_x$ into $\mathbf{\Sigma}_x \mathbf{u}_m = \lambda_m \mathbf{u}_m$ via EVD to extract the eigenvectors $\mathbf{U} = \{\mathbf{u}_m\}_{m=1}^{M}$ and the corresponding eigenvalues $\{\lambda_1, \lambda_2, \ldots, \lambda_M\}$.

The data $\mathbf{x}$ can now be visualized by linearly projection onto the 2 eigenvectors $\mathbf{U}_{2D} = \{\mathbf{u}_{max_1}, \mathbf{u}_{max_2}\}$ with the 2 largest eigenvalues to achieve the 2 dimensional variable $\mathbf{z}$ defined as

$$\mathbf{z} = \mathbf{U}_{2D}^T(\mathbf{x} - \mu_x) \tag{A.22}$$

The covariance of the lower dimensional data $\mathbf{\Sigma}_z$ can now be expressed based on the covariance $\mathbf{\Sigma}_x$ as

$$\begin{aligned}\mathbf{\Sigma}_z = \mathcal{E}\{\mathbf{z}\mathbf{z}^T\} &= \mathcal{E}\{\mathbf{U}_{2D}^T(\mathbf{x} - \mu_x)(\mathbf{x} - \mu_x)^T\mathbf{U}_{2D}\} \\ &= \mathcal{E}\{\mathbf{U}_{2D}^T\mathbf{x}\mathbf{x}^T\mathbf{U}_{2D} + \mathbf{U}_{2D}^T\mu_{\mathbf{x}}\mu_{\mathbf{x}}{}^T\mathbf{U}_{2D}\} \\ &= \mathbf{U}_{2D}^T\mathbf{\Sigma}_x\mathbf{U}_{2D} + \mathbf{U}_{2D}^T\mu_{\mathbf{x}}\mu_{\mathbf{x}}{}^T\mathbf{U}_{2D}\end{aligned}$$

where the last term on the right side is the projected mean. If we therefore assume $\mathbf{x}$ has zero mean $\mu_x = 0$ we can reduce the covariance $\mathbf{\Sigma}_z$ to

$$\mathbf{\Sigma}_z = \mathbf{U}_{2D}^T\mathbf{\Sigma}_{\mathbf{x}}\mathbf{U}_{2D} \tag{A.23}$$

This means the multivariate covariance $\mathbf{\Sigma}_x$ can now be reduced in dimensionality to $\mathbf{\Sigma}_z$ for easy viewing.

## A.5 Appendix - Extracting random sample from mixture models

In section 2.4 and 2.5 the Gaussian mixture model and the mixture of factor analyzers were presented. In this section we show how to extract a random variable $\mathbf{x}$ from each of these two mixture models given by their general form

$$p(\mathbf{x}) = \sum_k \alpha_k \cdot p(\mathbf{x}|k) \tag{A.24}$$

where $\alpha_k$ are the priors for each mixture component and $p(\mathbf{x}|k)$ denotes the kernel function. For the Gaussian mixture model $p(\mathbf{x}|k)$ is given by (2.30) and for the mixture of factor analyzers $p(\mathbf{x}|k)$ can be derived from (2.54), rewritten here for convenience

$$\text{MoG} \sim p(\mathbf{x}|k) = \mathcal{N}(\mu_k, \mathbf{\Sigma}_k) \tag{A.25}$$

$$\text{MFA} \sim p(\mathbf{x}|k) = \mathcal{N}(\mu_k, \mathbf{\Lambda}_k\mathbf{\Lambda}_k^T + \mathbf{\Psi}) \tag{A.26}$$

As mentioned earlier, the advantage of these mixture models are their ability to model complex probability distributions $p(\mathbf{x})$ through combinations of much simpler kernel functions $p(\mathbf{x}|k)$. Figure A.3 illustrates an example of such a mixture model with 4 mixture components.

Extracting a random variable $\mathbf{x}$ from equation (A.24) can be described as a two step procedure :

1. Select random mixture component $k$ with prior probability density $\alpha_k$.
2. Extract $\mathbf{x}$ from the $k$'th mixture component using the corresponding $k$'th set of parameters for the kernel function, $p(\mathbf{x}|k)$.

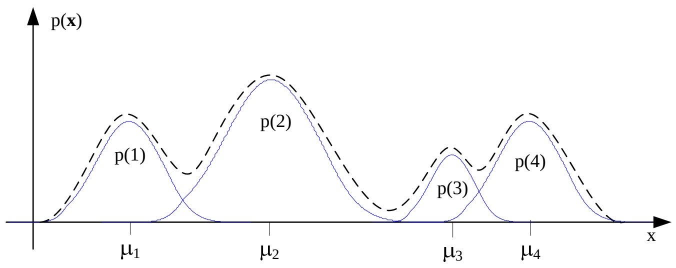

*Figure A.3:* Example of how $p(\mathbf{x})$ can be modelled by a mixture of 4 Gaussian kernel functions with individual mean vectors and covariance matrices as parameters.

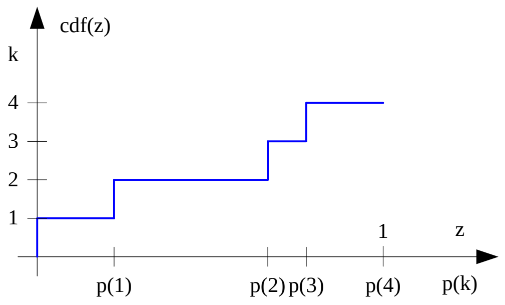

*Figure A.4:* Example showing how the $cdf(z)$ can be calculated by cumulating the probabilities in $\alpha_k$.

To extract a random mixture component $k$ we form the corresponding *cumulative distribution function* (cdf) from $\alpha_k$ by cumulating the probabilities $cdf(z) = \sum_{k=0}^{z} \alpha_k$, as illustrated in figure A.4. By generating a random variable $z$ from a uniform distribution, we can extract a random $k$ from $cdf(z)$ such that $k$ has $\alpha_k$ as probability density.

The data $\mathbf{x}$ is then generated by extracting from the kernel function using the $k$'th set of parameters, $\mathbf{\Theta}_k$. For the Gaussian kernel function given in (2.30) $\mathbf{\Theta}_k = \{\mu_k, \mathbf{\Sigma}_k\}$ and for the factor analyzer given in (2.53) $\mathbf{\Theta}_k = \{\mu_k, \mathbf{\Lambda}_k, \mathbf{\Psi}\}$.

However instead of using a Gaussian distribution directly for the MFA we can exploit the FA linear model $\mathbf{x} = \mathbf{\Lambda z} + \epsilon$ in the extraction. If we expand $p(\mathbf{x}|k)$ to

$$p(\mathbf{x}|k) = \int p(\mathbf{x}|\mathbf{z}, k) p(\mathbf{z}|k) \, \mathrm{d}\mathbf{z} \tag{A.27}$$

and identify $p(\mathbf{x}|\mathbf{z}, k) = \mathcal{N}(\mu_k + \mathbf{\Lambda}_k \mathbf{z}, \mathbf{\Psi})$ and $p(\mathbf{z}|k) = p(\mathbf{z}) = \mathcal{N}(0, \mathcal{I})$, we can lead to a more simplified procedure of extracting data. Initially a random factor $\mathbf{z}$ is generated from $\mathcal{N}(0, \mathcal{I})$ and mapped to an $d$-dimensional affine space by computing $\mu_k + \mathbf{\Lambda}_k \mathbf{z}$. Afterwards $\mathbf{x}$ is formed by adding $\epsilon$ generated from $\mathcal{N}(0, \mathbf{\Psi})$.

These procedures for extracting random variables from the Mog and MFA model are both implemented in `MATLAB` in appendix A.5.

## A.6 Appendix - Maximum Likelihood of Gaussian data

In this section we derive a maximum likelihood parameter estimate of a linear model with additive Gaussian noise and show how this is related to the least-squared estimate. We further show how the decomposition of the LS-estimate leads to the *Bias/Variance Tradeoff* and how this can be used in evaluating model complexity.

Consider an observed dataset $\mathbf{X} = \{\mathbf{x}_n\}_{n=1}^{N}$ with additive Gaussian noise $\epsilon$ distributed by $\mathcal{N}\{0, \sigma^2 \mathcal{I}\}$, where $\sigma^2$ is a vector with same dimension as $\mathbf{x}$. If we further assume $\mathbf{x}$ is generated from a linear composition, we can express the model of the observed data by

$$\mathbf{x} = \mathbf{A}\mathbf{s} + \epsilon \tag{A.28}$$

where $\mathbf{A}$ and $\mathbf{S}$ are both considered model parameters given by $\mathbf{A} = \{\mathbf{a}_r\}_{r=1}^{d}$ and $\mathbf{S} = \{\mathbf{s}_n\}_{n=1}^{N}$, where s is a $d$ dimensional source-vector. If we assume the noise $\epsilon$ is uncorrelated with isotropical variance (i.e. $\sigma$ is a scalar) we can express the likelihood of a single observed datasample as

$$p(\mathbf{x}|\mathbf{s}) = \frac{1}{\sqrt{(2\pi)^M |\sigma^2 \mathcal{I}|}} \exp\Big( - \frac{1}{2\sigma^2} (\mathbf{x} - \mathbf{A}\mathbf{s})^T (\mathbf{x} - \mathbf{A}\mathbf{s}) \Big) \tag{A.29}$$

The log-likelihood for all observed datasamples can now be written as

$$\mathcal{L} = \log p(\mathbf{X}|\mathbf{S}) = \sum_{n=1}^{N} \left[ -\frac{1}{2\sigma^2}(\mathbf{x}_n - \mathbf{A}\mathbf{s}_n)^T(\mathbf{x}_n - \mathbf{A}\mathbf{s}_n) \right] + \log(k) \tag{A.30}$$

$$= -\frac{1}{2\sigma^2} \parallel \mathbf{X} - \mathbf{AS} \parallel^2 + \log(k) \tag{A.31}$$

where $\log(k)$ is a normalization constant. In a max. likelihood estimate of the model parameters $\mathbf{A}$ and $\mathbf{S}$ we see that the maximum is reached when $\mathbf{X} = \mathbf{AS}$. More importantly eq. (A.31) shows that a max. likelihood estimate are the same as minimizing the least-squared error. This simple relation is the main assumption for employing the least-squared estimate as a max. likelihood solution, as we have done in our optimization of the matrix factorizations in sections 2.1, 4.3.1 and 4.3.2.

We can further expand the analysis by denoting the model estimate $\mathbf{As} + \epsilon$ as dependent on $\mathbf{x}$ by $y(\mathbf{x})$. If we now evaluate the expectation of the least-squared estimate [4] we get

$$E = \mathcal{E}\Big\{\big[y(\mathbf{x}) - \mathcal{E}\{\mathbf{x}\}\big]^2\Big\} = \underbrace{\Big[\mathcal{E}\{y(\mathbf{x})\} - \mathcal{E}\{\mathbf{x}\}\Big]^2}_{(Bias)^2} + \underbrace{\mathcal{E}\Big\{\big[y(\mathbf{x}) - \mathcal{E}\{y(\mathbf{x})\}\big]^2\Big\}}_{Variance} \tag{A.32}$$

This shows how the generalization error can be expanded into a *bias* and a *variance* term. In evaluating the complexity of a given model, these two terms become useful. If a model estimate is on average different over all datasets from the desired function $\mathcal{E}\{\mathbf{x}\}$, then it is said to suffer from a bias and manifests in a large bias term. In contrast if the model estimate is sensitive to a particular training set it is considered overfitted and will have a large variance term in (A.32). In case of a perfect optimization, where $\mathbf{X} = \mathbf{AS}$, we see that the bias term disappears and the variance term is reduced to the variance of the noise $\epsilon$.

## A.7 Appendix - Gibbs sampling

Extracting samples from a complex probability distribution $p(\mathbf{x}, c)$ without a simple analytic expression can be achieved by the iterative *Gibbs sampling* method. The joint probability can be factorized into two different conditional distribution as $p(\mathbf{x}, c) = p(\mathbf{x}|c)p(c) = p(c|\mathbf{x})p(\mathbf{x})$. To conduct the sampling these conditional probabilities must be tractable as they form the iterative steps in the algorithm.

Gibbs sampling is illustrated in figure A.5 for the two dimensional case $p(\mathbf{x}) = p(x_1, x_2)$ as an example, where each iteration consists of two similar steps. For the current state $\mathbf{x}^{(t)}$, $x_2^{(t)}$ is used to extract the next sample $x_1^{(t+1)}$ from the conditional distribution $p(x_1^{(t+1)}|x_2^{(t)})$ and similarly for $x_2^{(t+1)}$ forming the next state $\mathbf{x}^{(t+1)}$.

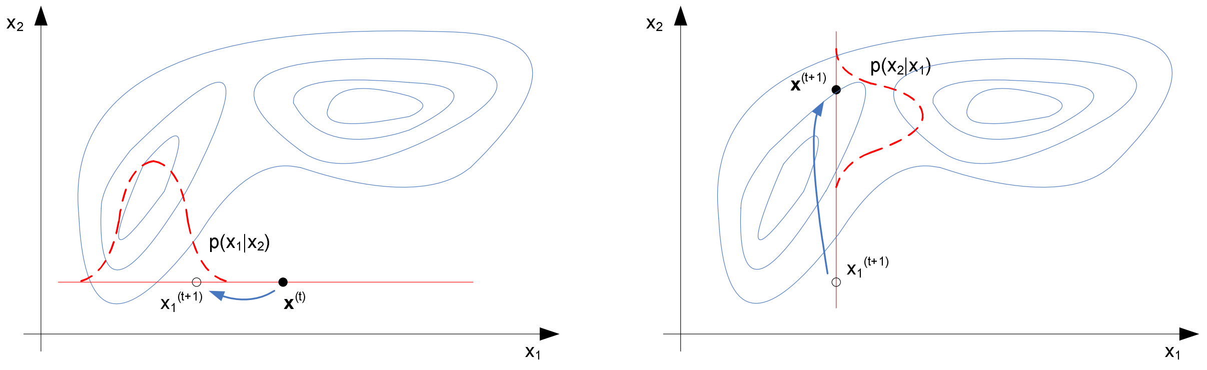

*Figure A.5:* Principal illustration of Gibbs sampling from a joint distribution $p(x_1, x_2)$

These steps form an iterative algorithm for which it can be shown that $\mathbf{x}^{(t)}$ tends to $p(\mathbf{x})$ for $t \rightarrow \infty$ [19]. This also makes intuitive sense from the figure A.5, where the iterations ensures settlement 'within' the distribution.

# B. Appendix: Code Section

In this section the MATLAB code is listed in all its detail. Certain functions are part of a tool-package and is not listed here, please refer to the DVD for a reference.

## B.1 Appendix - General MATLAB code

Show image patches

```
function showPatch(D, K, scale, L)

% Show image patches
% -------------------------------------------------------------------------
% Input parameters
% D                  : Data as a d x N matrix
% K                  : No. of patches in a row, width of figure
% scale              : 'single' - Each patch is preprocessed for max.
%                       constrast, i.e. dyn. range of [0:1]
%                      'all' - All patches are scaled as one to have
%                       better view energy
%                      'none' - Truncates the negative part due to uint8
% L                  : Vector holding borders for vertical lines, used
%                      with K-means.
%
% Output parameters
% Figure illustrating image patches.
% -------------------------------------------------------------------------

% Initalize
if nargin < 4  L = 0; end
d = 3;

% Dimensions
dY = sqrt(size(D,1));
dX = sqrt(size(D,1));

k = 1; dx = d; dy = d;
N = size(D,2);
Y = ceil(N/K);

% Initialize with white background
View  = 255*ones(dY*Y+d*(Y+1), dX*K+d*(K+1), 'uint8');

switch (scale)
  case 'single'
    % Pre-process - Scale individual patch
    D = D  - repmat(min(D), size(D,1), 1);           % Add min. to have non-neg. values
    D = D ./ repmat(max(D)+eps,  size(D,1), 1);      % Scale for max. contrast
  case 'all'
    D = D ./ repmat(max(abs(D))+eps,  size(D,1), 1)+0.5;     % Scale for max. contrast
end

% Convert to uint8 for plotting
D = uint8(255*(1-D));

% Convert vectors to image patches
```

```
for k = 1:N
  xPos = (dX+d)*mod(k-1, K)+1  + d;
  yPos = (dY+d)*floor((k-1)/K) + d;

  patch = reshape(D(:,k), dX, dY, 1);
  View(yPos:yPos+dY-1, xPos:xPos+dX-1) = patch;
end

% Finally show patches as image
View(1,:) = 0; View(end,:) = 0; View(:,1) = 0; View(:,end) = 0;     % Black border

% Insert horizontal lines
if (length(L) > 1)
  for i = 0:(N/K)-1
    View(i*31+1,:) = 0;
  end
end

% Insert vertical lines for grouping
j = 0;
for i = 1:length(L)-1
  j = j + L(i);
  x = mod(j,K) * (dX+3) + d+1;
  y = floor(j/K);
  View(y*(dX+3)+1:(y+1)*(dX+3)+1, x) = 1;
  imshow(View);
end

% Finally show figure
imshow(View);
```

## B.2 Appendix - MATLAB code for Mixture Models

MATLAB code - The K-Means algorithm

```
function [CC, CID, dist_2_CC , it] = K_means(D, K, CC, opt)

% K-means
% --------------------------------------------------------------------------
% Input parameters
% D               : Data as column vectors
% K               : Amount of clusters
% CC              : Initial cluster centers
% opt.plot        : Enable plotting during iteration, employes PCA

% Output parameters
% CC              : Resulting cluster centers
% CID             : Cluster assignments
% dist_2_CC       : Metric distance to assigned cluster center
% it              : No. of iterations used.
% --------------------------------------------------------------------------


fprintf(['K-Means,␣K␣=␣' num2str(K) '␣clusters...']);

% Prepare data if plotting flag enabled
if (opt.plot == 2)
  % PCA to visualize
  D_mean   = mean(D,2);
  D_m      = D - repmat(D_mean, 1, size(D,2));      % Subtract mean

  % Calc. eigenvectors
  cov_D   = D_m*D_m'/size(D_m,2);
  [E, L] = eig(cov_D);
  U       = E(:,end:-1:1);                   % Reverse for largest eigenvector first

  % Project data onto largest principal components
  D_pca = U(:,1:2)' * D_m;                   % Reduce dimensions
  clear D_m cov_D E;

  CC_init = U(:,1:2)' * (CC - repmat(D_mean, 1, size(CC,2)));
end

% Initialize
N = size(D,2);  % number of training examples
CID = zeros(N,1);       CID_old = ones(N,1);    it = 1;

while ~isequal(CID, CID_old) & it < opt.maxIT
  CID_old = CID;

  % Calc. distance as (a+b)^2
  dist = ones(N,1)*sum(CC.*CC) + sum((D.*D))'*ones(1,K) - 2*D'*CC;

  % Find index of closest cluster
  [dist_2_CC, CID] = min(dist,[],2);
  if (K == 1)
    dist_2_CC = dist; CID = ones(1,N);       % dummy if K = 1
  end

  % Re-estimate the cluster centers
  for k = 1:K
    indexk = find(CID==k);
    if sum(indexk) > 0,
      CC(:,k) = mean(D(:,indexk), 2);
    end
  end
  it = it + 1;
  fprintf('.');

  if (opt.plot == 1)
    % Plot
    MU = U(:,1:2)' * (CC - repmat(D_mean, 1, size(CC,2)));

    figure(200), plot(D_pca(1,:), D_pca(2,:) , '*g'), hold on;
    plot(MU(1,:), MU(2,:), 'or', 'LineWidth', 3);
    plot(CC_init(1,:), CC_init(2,:), '*b');
    xlabel('1.␣Principal␣Componens'); ylabel('2.␣Principal␣Component');
    hold off, grid, drawnow;
  end
end
it-1;

fprintf('Ok\n');
return;
```

MATLAB code - Codebook Clustering

```
% Codebook clustering using K-Means algorithm

clear; close all;

% Initialize
digit        = 2;                % Which digit
d            = 30;               % No. of NMF features, dimensions
K            = [5 8 10 15 20];   % Mixture components
opt.maxIT    = 100;              % Stop criteria
opt.plot     = 0;                % Plot during runtime

% Load data
load (['Data/MNIST/Code/NMF_pp/nmf_' num2str(digit) '_d' num2str(d) '.mat']);

[d N] = size(A);
D     = zeros(d,length(K)*10);

for kk = 1:length(K)
  % Initialize with random datapoint as clusters.
  CC = A(:, ceil(N*rand(K(kk),1)));

  % Run K-means
  [mu, CID, dist_2_CC, it] = K_means(A, K(kk), CC, opt);

  % Sort by Cluster ID
  A_ = ones(d, N); i = 1;
  for k = 1:K(kk)
    id = find(CID == k); lid(k) = length(id);
    if (lid(k) ~= 0)
      A_(:,i:i+lid(k)-1) = A(:,id);
      i = i+lid(k);
    end
  end

  % Extract from model
  D_temp = zeros(d,10);
  for j = 1:10
    i = 1;
    for k = 1:K
      if (lid(k) ~= 0)
        seg = A_(:,i:i+lid(k)-1);
        i = i+lid(k);
        D_temp(:,j) = D_temp(:,j) + seg(:, ceil(lid(k)*rand(1)));
      end
    end
  end

  D(:,(kk-1)*10+1:(kk*10)) = D_temp;
end

% Plot
figure(1), showPatch(mu, 10, 'single');
figure(3), showPatch(A_, 10, 'single', lid);
figure(10), showPatch(D, 10, 'single');
```

MATLAB code - Mixture of Gaussians

```
function [mu sigma2 p_k_x p_k perf] = MoG(D, K, opt);

% Mixture of Gaussians used the EM algorithm
% ------------------------------------------------------------------------
% Input parameters
% D                : Data as a d x N matrix
% K                : Number of clusters
% opt.variance     : Type of covariance matrix sigma2
%                    'full' - full d x d covariance matrix
%                    'diag' - diagonal matrix with sigma2 defined as d x K
%                             matrix
%                    'iso'  - isotropical variance with a constant down the
%                             diagonal of sigma2 defined as a K dim. vector.
% opt.resp_tol     : Log-likelihood tolerance use as stopping criteria
% opt.maxIT        : Max. number of EM iterations
% opt.plot         : Plot during iterations
% opt.debug        : Debug code by forcing 2D data, i.e. no PCA applied.

% Output parameters
% mu               : Means of clusters as a d x K matrix
% sigma2           : Covariances of clusters, see opt.variance above for
%                    formats
% p_k_x            : Probability p(k|x) given as a K x N matrix
% p_k              : Probability p(k) given as a K dim. vector
% perf.it          : Number of iterations used
% perf.ll          : Log-likelihoods during iterations
% perf.resp_delta : Difference in responsibility
% ------------------------------------------------------------------------


% Force 2 dim. data during debuging
if (opt.debug == 1)
  D = D(1:2,:);
end

% Initialize
fprintf(['Gaussian␣Mixture␣Model,␣K␣=␣' num2str(K) '␣components...']);
[d N]   = size(D);

% Prepare data if plotting flag enabled
if (opt.plot == 1)
  if opt.debug == 0
    % PCA to visualize
    D_mean    = mean(D,2);
    D_m       = D - repmat(D_mean, 1, size(D,2));    % Subtract mean

    % Calc. eigenvectors
    cov_D   = D_m*D_m'/size(D_m,2);
    [E, L] = eig(cov_D);
    U       = E(:,end:-1:1);                     % Reverse for largest eigenvector first

    % Project data onto largest principal components
    D_pca = U(:,1:2)' * D_m;                     % Reduce dimensions
    clear D_m cov_D E;
  else
    % if debugging, use dummy transformation matrix etc.
    U = eye(2); D_mean = zeros(d,1);
    D_pca = D;
  end
end
```

```
% Initialize model parameters
logp_k     = log(ones(K,1)/K);                          % Equal weighting of clusters
p_x_old    = 0;
ll         = 0;

% Select random datapoints as initial means
mu = D(:, ceil(N*rand(K,1)));

% Initialize variances with covar matrix for entire dataset.
temp      = D - repmat(mean(D,2), 1, N);
temp      = temp*temp' / N + 1e-6*eye(d);

switch (opt.variance)
  case 'full'
    sigma2 = repmat(temp, [1 1 K]);               % Use entire matrix
  case 'diag'
    sigma2 = repmat(diag(temp), [1 K]);           % Extract only diagonal
  case 'iso'
    sigma2 = trace(temp)*rand(K,1) / N;           % Use sum of diagonal (trace)
    sigma2 = rand(K,1);

    % Square input data for later use
    D_2 = ones(K,1)*sum((D.*D));
end
clear temp;


% Expectation-Maximization loop
% -------------------------------------------------------------------------

% Init parameters.
mu_old = mu; it = 1; resp_delta = inf; p_k_x_old = zeros(K,N);
while (it < opt.maxIT & resp_delta > opt.resp_tol)

  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  % Expectation step

  % Calc. log(p_k_x)
  for k = 1:K
    % Extract mean for the k'th cluster
    mu_k = mu(:,k); mu_k = mu_k(:,ones(1,N));

    switch (opt.variance)
      case 'full'
        i_sigma2       = inv(squeeze(sigma2(:,:,k)));
        Gauss_norm     = - 0.5*( d*log(2*pi) + logdet(squeeze(sigma2(:,:,k))) );
        logp_x_k(k,:) = logp_k(k) + Gauss_norm ...
                          - 0.5 * (sum( (D-mu_k).*(i_sigma2*(D-mu_k)), 1 ));
      case 'diag'
        i_sigma2       = 1 ./ max(eps, sigma2(:,k));
        Gauss_norm     = - 0.5*( d*log(2*pi) + sum(log(sigma2(:,k)) ));
        logp_x_k(k,:) = logp_k(k) + Gauss_norm - 0.5 * i_sigma2' * ( (D-mu_k).^2 );
      case 'iso'
        i_sigma2       = 1./sigma2(k); i_sigma2 = i_sigma2(ones(d,1), :);
        Gauss_norm     = - d/2 * log(2*pi*sigma2(k));
        logp_x_k(k,:) = logp_k(k) + Gauss_norm - 0.5 * i_sigma2' * ( (D-mu_k).^2 );
    end

  end
  max_logp_x_k = max(logp_x_k);            % numerical stability, remove largest value
  p_x_k        = exp(logp_x_k - max_logp_x_k(ones(K,1),:));
  p_x          = sum(p_x_k,1);
  p_k_x        = p_x_k ./ p_x(ones(1,K), :);        % Responsibility
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Maximization step

% Mean
mu   = ( diag(1./sum(p_k_x')) * p_k_x * D' )';

% Variance
switch (opt.variance)
  case 'full'
    for k = 1:K
      mu_k            = mu(:,k); mu_k = mu_k(:,ones(1,N));
      p_k_x_k         = p_k_x(k,:); p_k_x_k = p_k_x_k(ones(d,1), :);
      sigma2(:,:,k) = ((D - mu_k) * (p_k_x_k' .* (D - mu_k)')) / sum(p_k_x(k,:));
      sigma2(:,:,k) = sigma2(:,:,k) + 1e-6*eye(d); % Numerical stability during inv.
    end
  case 'diag'
    for k = 1:K
      mu_k          = mu(:,k); mu_k = mu_k(:,ones(1,N));
      p_k_x_k       = p_k_x(k,:); p_k_x_k = p_k_x_k(ones(d,1), :);
      sigma2(:,k) = sum(p_k_x_k.*(D-mu_k).^2,2) / sum(p_k_x(k,:));
    end
  case 'iso'
    dist    = sum(mu.*mu)'*ones(1,N) + D_2 - 2*mu'*D;
    sigma2 = (1/d)*diag(1./sum(p_k_x')) * (sum((dist.*p_k_x)')') + 1e-6*ones(K,1);
end

% Component prior
p_k = sum(p_k_x, 2) / N;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Log-likelihood

% Stopping criteria - Responsibility
ll(it+1) = sum(log(sum(exp(logp_x_k))));

% Compute change in responsibility
resp_delta             = sum(sum(abs(p_k_x_old - p_k_x))) / N;
p_k_x_old              = p_k_x;
perf.resp_delta(it) = resp_delta;

it = it + 1;
fprintf('.');

if (opt.plot == 1)
  % Plot, slow!
  MU = U(:,1:2)' * (mu - repmat(D_mean, 1, size(mu,2)));

  figure(200), plot(D_pca(1,:), D_pca(2,:) , '.g'), hold on;
  plot(MU(1,:), MU(2,:), 'or', 'LineWidth', 3);
  for k = 1:K
    switch (opt.variance)

      % Full, draw ellipses
      case 'full'
        circle = [cos(2*pi*(1:101)/100); sin(2*pi*(1:101)/100)];
        if (opt.debug == 1)
          [E, Lambda] = eig(sigma2(:,:,k));
          x             = E*diag(sqrt(diag(Lambda))) * circle;
        else
          e = U; e = e(:,1:2);
          x = e'*(sign(sigma2(:,:,k)).*sqrt(abs(sigma2(:,:,k))))*e * circle;
          end
          plot(x(1,:)+MU(1,k), x(2,:)+MU(2,k), 'r', 'LineWidth', 2);

        % Diagonal, draw ellipses
        case 'diag'
          circle = [cos(2*pi*(1:101)/100); sin(2*pi*(1:101)/100)];
          if (opt.debug == 1)
            [E, Lambda] = eig(diag(sigma2(:,k)));
            x           = E*diag(sqrt(diag(Lambda))) * circle;
          else
            e = U; e = e(:,1:2);
            x = e'*diag(sqrt(sigma2(:,k)))*e * circle;
          end
          plot(x(1,:)+MU(1,k), x(2,:)+MU(2,k), 'r', 'LineWidth', 2);

        % Isotropical, draw circles
        case 'iso'
          plot(MU(1,k)+sqrt(sigma2(k))*sin(2*pi*(0:31)/30), ...
               MU(2,k)+sqrt(sigma2(k))*cos(2*pi*(0:31)/30),'r', 'LineWidth', 2);
      end
    end
    xlabel('1.␣Principal␣Componens'); ylabel('2.␣Principal␣Component');
    hold off, grid, axis equal, drawnow;
  end
end

% Return figures
perf.it = it;
perf.ll = ll(2:end);
fprintf('Ok\n');

function y = logdet(A)
% log(det(A)) where A is positive-definite.
% This is faster and more stable than using log(det(A)).
% U = chol(A);
% y = 2*sum(log(diag(U)));
% Alternativ by Thomas Grotkjær

[U,p] = chol(A);
if ~p
  y = 2*sum(log(diag(U)));
else
  y = 1;
end
```

MATLAB code - Extract from MoG Model

```
function [s k] = MoG_extract(P_k, mu, sigma2, covar, N);


% Extract samples from a mixture of Guassians
% -------------------------------------------------------------------------
% Input parameters
% P_k            : Priors
% mu             : Cluster means
% sigma2         : Covariance matrix
% covar          : Type of covariance, 'iso', 'diag' or 'full'.
% N              : No. of samples to generate
%
% Output parameters
% s              : N samples as column vectors
% k              : Component membership for each sample
% -------------------------------------------------------------------------


% Initialize
[d K] = size(mu);

% Calc. CDF from P(k) probabilities
cdf_k = P_k(1);
for i=2:K cdf_k(i)=cdf_k(i-1)+P_k(i); end

% Generate N samples
for i = 1:N
  k(i)     = length(find(rand(1)>cdf_k))+1;      % Extract random cluster
  s_m(:,i) = mu(:,k(i));

  % Extract from k'th components depending on type
  switch (covar)
    case 'full'
      s(:,i) = (randn(1,d) * chol(sigma2(:,:,k(i))) )' + mu(:,k(i));
    case 'diag'
      s(:,i) = randn(d,1) .* sqrt(sigma2(:,k(i))) + mu(:,k(i));
    case 'iso'
      s(:,i) = randn(d,1) .* sqrt(sigma2(k(i))) + mu(:,k(i));
  end
end
```

MATLAB code - Extract from MFA Model

```
function [s k] = MFA_extract(P_k, Lh, psi, mu, N)

% Extract samples from a mixture analyzer
% -------------------------------------------------------------------------
% Input parameters
% P_k            : Priors
% Lh             : Factor loading matrix
% psi            : Vector for diagonal matrix Psi
% mu             : Cluster means
% N              : No. of samples to generate

% Output parameters
% s              : N samples as column vectors
% k              : Component membership for each sample
% -------------------------------------------------------------------------


[d dz K] = size(Lh);

% Calc. CDF from P(k) probabilities
cdf_k = P_k(1);
for i=2:K cdf_k(i)=cdf_k(i-1)+P_k(i); end

% Generate random z
z   = randn(dz,N);
eps = diag(psi)*randn(d,N);

% Generate N samples
for i = 1:N
  k(i)   = length(find(rand(1)>cdf_k))+1;                % Extract random cluster
  s(:,i) = mu(:,k(i)) + Lh(:,:,k(i))*z(:,i) + eps(:,i);
end
```

## B.3 Appendix - MATLAB code for Deep Networks

MATLAB code - Single layer network (training, testing, extracting features).

```
function [A, S, W, C, opt, err_c] = NLMF(X, c, d, opt)

% NLMF: NON-LINEAR MATRIX FACTORIZATION
%
% Authors:
%   Morten Arngren
%   Technical University of Denmark,
%   Institute for Matematical Modelling
%
% Usage:
%   [A S C] = NLMF(X, d, opt)
%
% Input:
%   X                  M x N data matrix
%   d                  Number of factors
%   opt
%   .type              Model type:
%                      - 'uncon'  for unconstrained model
%                      - 'nonneg' for non-negative model
%   .fs                Mapping function:
%                      - 'un2un' f_S = tanh(S);
%                      - 'nn2un' f_S = exp(S) ./ (1 + exp(S));
%                      - 'nn2nn' f_S = S ./ (1+S);
%   .softmax           Incl. classifier
%   .C                 No. of classes
%   .alpha             Reg. of sources, s
%   .beta              Reg. of features, A
%   .gamma             Reg. of class. weights, W
%   .sigma2            Balance parameter
%   .A                 Codebook matrix A as input
%   .S                 Sources as input
%   .W                 Class. weight as input
%
%   .cost_delta        Stop criteria, diff. between cost iterations
%   .IT_out            Max. no. of iteration, outer loop
%   .IT_in             MAx. no. of iteratoins, inner loops
%   .A_cost_delta      Stop criteria, for A, inner loop
%   .S_cost_delta      Stop criteria, for S, inner loop
%   .W_cost_delta      Stop criteria, for W, inner loop
%
%   .loop              Loop inside A,S & W to converge
%   .test              Test session, learn only S
%   .class             Run classifier only, for dual sessions
%   .plot              Plot during learning, slow!

% Output:
%   A                  Features,         M x (d+1) matrix
%   S                  Codevectors,      (d+1) x N matrix
%   W                  Class. weights,   C x (d+1) matrix
%   C                  Cost function values
% -------------------------------------------------------------------------

% Setup arguments if activating class. network only.
if opt.class
  opt.softmax       = 1;
  opt.test          = 0;
  opt.loop          = 1;
  opt.W_cost_delta = opt.cost_delta;    % Use S stop. criteria as global
  opt.IT_in         = opt.IT_out;       % Use inner maxIT as global
  opt.A             = 0;
end

% Setup arguments if using testset
if opt.test
  opt.softmax       = 0;                 % Don't optimize W
  opt.loop          = 1;
  opt.S_cost_delta  = opt.cost_delta;    % Use S stop. criteria as global
  opt.IT_in         = opt.IT_out;        % Use inner maxIT as global
  opt.sigma2        = 1;
end

fprintf('\n');
fprintf('Non-Linear␣Matrix␣Factorization\n');
fprintf('========================================\n');

if strcmp(opt.type, 'nonneg') && ~strcmp(opt.fs, 'nn2nn')
  fprintf('-␣Non-linear␣mapping␣function␣f(s)␣forced␣to␣nn2nn\n');
  opt.fs = 'nn2nn';
end
if strcmp(opt.type, 'uncon') && strcmp(opt.fs, 'nn2nn')
  fprintf('-␣Non-linear␣mapping␣function␣f(s)␣forced␣to␣un2nn\n');
  opt.fs = 'un2nn';
end

[M N] = size(X);

% Display information
if ~opt.test
  fprintf('-␣Training␣on␣%5.0f␣MNIST␣samples␣using␣%s␣constraint.\n\n', N, opt.type);
else
  fprintf('-␣Testing␣on␣%5.0f␣MNIST␣samples␣using␣%s␣constraint.\n\n', N, opt.type);
end
fprintf('%12s␣|␣%12s␣|␣%12s␣|␣%12s␣|␣%12s␣|␣%12s␣|␣%6s\n','Iteration','NuA', ...
        'NuS','NuW','Cost␣func.','Delta␣costf.','Err_C␣[%]');
fprintf('-------------+--------------+--------------+--------------+-----------\n');

rand('state',sum(100*clock));            % See random generator

% Initialize model parameters A, W and S
if ~opt.class
  S = [1*(rand(d,N)); ones(1,N)];
else S = opt.S; end

if ~opt.test
  % A = [ 0.1*X(:,ceil(N*rand(1,d))) zeros(M,1)];   % Init. with digits
  A = 0.1*[rand(M,d) zeros(M,1)];                   % Init. with noise
  W = rand(opt.C, d+1);
  if strcmp(opt.type, 'uncon')
    A             = A - ones(M,1)*sum(A,1)/size(A,1);
    S(1:end-1,:)  = S(1:end-1,:) - 0.5;
    W             = W - 0.5;
  end
else
  A = opt.A; W = opt.W;
end

% Calc. non-linear sources for first iteration
f_S  = mapS(S, opt);

% Initialize algorithm parameters
switch opt.type
  case 'nonneg' % Multiplicative stepsizes
```

```
    opt.nuA = 1; opt.nuS = 1; opt.nuW = 1;
    if opt.class opt.nuW = 1e-3; end
  case 'uncon'  % Additive stepsizes
    opt.nuA  = 1e-3;
    opt.nuS  = 0.01;
    opt.nuW  = 0.01;
    if opt.test opt.nuS = 1e-2; end
end
opt.accel = 1.3;
cost.E = inf; cost.Ec = inf; cost.Ed = inf; cost_old = inf; delta_cost = inf;
it = 0; err_c = inf;

% Select random digits for viewing during iterations
if opt.plot
  id = ceil(N*rand(1,20));
end


% Optimization loop
while (it < opt.IT_out && (delta_cost) >= cost.E*opt.cost_delta)

  % Optimize parameters
  if (~opt.test && ~opt.class)
    [A, cost, opt]      = Optmize_A(A, X, S, f_S, cost, opt); end

  if (~opt.test && opt.softmax)
    [W, cost, opt]      = Optmize_W(W, f_S, c, d, cost, opt); end

  if (~opt.class)
    [S, f_S, cost, opt] = Optimize_S(S, A, W, X, c, d, cost, opt); end

  if ~opt.test
    delta_cost = cost_old - cost.E;
  else
    delta_cost = 0;                 % Run outer iteration once for test-error
  end
  cost_old     = cost.E;
  it           = it + 1;
  C(it)        = cost;
  [P_c err_c] = Softmax(W, S, c, opt);

  % Display information
  fprintf('%12.0f␣|␣%12.6g␣|␣%12.6g␣|␣%12.6g␣|␣␣%11.2f␣|␣␣%11.2f␣|␣␣%4.4f␣\n', ...
    it,opt.nuA/opt.accel,opt.nuS/opt.accel,opt.nuW/opt.accel,cost.E, ...
    delta_cost, err_c*100);

  if opt.plot
    view = [A(:,1:min(d,9)) A(:,d+1) A*f_S(:,id)];
    figure(1), showPatch(view, 10, 'single'); drawnow;
    figure(2), showPatch(A, 20, 'all'); title('Columns␣of␣A');
  end
end
[P_c err_c] = Softmax(W, S, c, opt);
return


% ----------------------------------------------------------------------
function [A, cost, opt] = Optmize_A(A, X, S, f_S, cost, opt)

A_old = A;
loop  = 1;

while (loop)
```

```
  cost_old = cost; it = 0;

  switch opt.type
    case 'nonneg'
      % Multiplicative update of A
      A_ = (X*f_S') ./ (A*f_S*f_S' +eps);       % Gradient
    case 'uncon'
      % Additive update of A
      A_ = (A*f_S - X)*f_S';                    % Gradient
      A_ = A_ + opt.beta*sign(A);               % Reg. term
  end

  while (1)
    switch opt.type
      case 'nonneg' % Multiplicative update
        A = A_old .* (A_.^opt.nuA);             % Take step
      case 'uncon'  % Additive update
        A = A_old - opt.nuA*A_;                 % Take step
    end

    cost.Ec = .5/opt.sigma2*norm(X - A*f_S,'fro')^2;      % Calc. energy as cost
    cost.Ec = cost.Ec + opt.alpha*norm(S,1) + opt.beta*norm(A,1);

    if cost.Ec < cost_old.Ec
      if (cost_old.Ec-cost.Ec) < cost.Ec*opt.A_cost_delta   % Stop if converged
        loop = 0;
      else loop = opt.loop; end
      A_old = A;
      opt.nuA = opt.nuA*opt.accel; break;       % Increase stepsize and exit
    else
      opt.nuA = opt.nuA/2 + eps;
      it = it + 1;
      if it > opt.IT_in                         % Exit if not converged
        loop = 0;
        A    = A_old;                           % Revert to old A and cost
        cost = cost_old;
        break;
      end
    end
  end
end
cost.E = cost.Ec + cost.Ed;
return

% -----------------------------------------------------------------------
function [S, f_S, cost, opt] = Optimize_S(S, A, W, X, c, d, cost, opt)

C     = size(W,1);
S_old = S;
loop  = 1;

while (loop)
  cost_old = cost; it = 0;

  % Map S through non-linear function f(s)
  f_S  = mapS(S, opt);
  f__S = mapS_(S, f_S, opt);            % Derivative of f(s)

  switch opt.type
    case 'nonneg' % Multiplicative update
      if ~opt.softmax
        S_ = (A'*X) ./ (A'*A*f_S + opt.alpha*opt.sigma2 +eps);        % Gradient
      else
```

```
        exps = exp(W*f_S);
        y    = exps ./ (ones(C,1)*sum(exps));
        S_   = A'*X/opt.sigma2 + W(c+1,:)';
        S_   = S_ ./ ( A'*A*f_S/opt.sigma2 +  W'*y + opt.alpha./(f__S+eps));
      end
    case 'uncon'  % Additive update
      S_      = f__S .* (A'*A*f_S - A'*X);                    % Gradient
      S_      = S_ + opt.alpha*opt.sigma2*sign(S);            % Reg. term
      if opt.softmax                                          % Softmax
        exps = exp(W*f_S);
        S_   = S_ + ( ( W'*exps ./ (ones(d+1,1)*sum(exps)) - W(c+1,:)' ) ...
                    .* f__S )*opt.sigma2;
      end
  end

  % Inner iterating loop
  while (1)
    switch opt.type
      case 'nonneg' % Multiplicative update
        S(1:d,:) = S_old(1:d,:) .* (S_(1:d,:).^opt.nuS);    % Take step
      case 'uncon'  % Additive update
        S(1:d,:) = S_old(1:d,:) - opt.nuS*S_(1:d,:);        % Take step
    end

    f_S          = mapS(S, opt);
    cost.Ec      = .5/opt.sigma2*norm(X - A*f_S,'fro')^2;  % Calc. energy as cost
    cost.Ec      = cost.Ec + opt.alpha*norm(S,1) + opt.beta*norm(A,1);
    if opt.softmax
      cost.Ed    = - sum(sum(W(c+1,:)'.*f_S)) + sum(log(sum(exp(W*f_S))));
      cost.Ed    = cost.Ed + opt.gamma*norm(W,1);
    else cost.Ed = 0;
    end

    cost.E       = cost.Ec + cost.Ed;
    if cost.E < cost_old.E
      if (cost_old.E-cost.E) < cost.E*opt.S_cost_delta  % Stop if converged
        loop = 0;
      else loop = opt.loop; end
      S_old = S;
      opt.nuS = opt.nuS*opt.accel; break;               % Increase stepsize and exit
    else
      opt.nuS = opt.nuS/2 + eps;
      it      = it + 1;
      if it > opt.IT_in                                 % Exit if not converged
        loop  = 0;
        S     = S_old;                                  % Revert to old S and cost
        cost  = cost_old;
        break;
      end
    end
  end

  % Display information
  if opt.test
    [P_c err_c] = Softmax(W, S, c, opt);
    fprintf('%12.0f␣|␣%12.6g␣|␣␣%11.2f␣|␣␣%4.4f␣\n', it, opt.nuS/opt.accel, ...
             cost.E, err_c);
  end

end
return
```

```
% ----------------------------------------------------------------------
function [W, cost, opt] = Optmize_W(W, f_S, c, d, cost, opt)

C      = size(W,1);
N      = size(f_S,2);
W_old = W;
loop   = 1;

while(loop)
  cost_old = cost; it = 0;

  exps   = exp(W*f_S);
  y      = exps ./ (ones(C,1)*sum(exps));
  switch opt.type
    case 'nonneg' % Multiplicative update
      for i = 1:opt.C
        c_id     = find(c==i-1);
        W_(i,:) = sum(f_S(:,c_id),2)';
        W_(i,:) = W_(i,:) ./ ( sum( (ones(d+1,1)*y(i,c_id)) .* f_S(:,c_id) ,2)' ...
                  + opt.gamma*ones(1,d+1));
      end
    case 'uncon'  % Additive update of W
      for i = 1:opt.C
        tcfs(i,:) = sum(f_S(:,find(c==i-1)),2)';
      end
      W_  = - tcfs + y*f_S' + opt.gamma*sign(W);
  end

  while (1)
    switch opt.type
      case 'nonneg' % Multiplicative update
        W = W_old .* (W_.^opt.nuW);                  % Take step
      case 'uncon'  % Additive update
        W = W_old - opt.nuW*W_;                      % Take step
    end
    cost.Ed = - sum(sum(W(c+1,:)'.*f_S)) + sum(log(sum(exp(W*f_S))));
    cost.Ed = cost.Ed + opt.gamma*norm(W,1);

    if cost.Ed < cost_old.Ed
      if (cost_old.Ed-cost.Ed) < cost.Ed*opt.W_cost_delta    % Stop if converged
        loop = 0;
      else loop = opt.loop; end
      W_old = W;
      opt.nuW = opt.nuW*opt.accel; break;            % Increase stepsize and exit
    else
      opt.nuW = opt.nuW/2 + eps;
      it = it + 1;
      if it > opt.IT_in                              % Exit if not converged
        loop = 0;
        W    = W_old;                                % Revert to old W and cost
        cost = cost_old;
        break;
      end
    end
  end
end
cost.E = cost.Ec + cost.Ed;

return
```

MATLAB code - Non-linear mapping function, f(s).

```
function [f_S] = mapS(S, opt)

% Non-linear mapping function f(s)
% ------------------------------------------------------------------------
% Input parameters
% S                : Sources
% opt.type         : Type of non-linear mapping function
%
% Output parameters
% f_S              : Mapped sources
% ------------------------------------------------------------------------

switch opt.fs
  case 'nn2nn'
    f_S = S ./ (1+S);
  case 'un2nn'
    f_S = exp(S) ./ (1 + exp(S));
  case 'un2un'
    f_S = tanh(S);
end

return
```

MATLAB code - Non-linear mapping function, 1st derivative, f'(s).

```
function [f__S] = mapS_(S, f_S, opt)

% 1st order derivatives of non-linear mapping function f(s)
% ------------------------------------------------------------------------
% Input parameters
% S                : Sources
% f_S              : Mapped sources, f(s)
% opt.type         : Type of non-linear mapping function
%
% Output parameters
% f__S             : 1st derivative of mapped sources
% ------------------------------------------------------------------------

switch opt.fs
  case 'nn2nn'
    f__S = (1 - f_S) ./ (S + 1);
  case 'un2nn'
    f__S = f_S - f_S.^2;
  case 'un2un'
    f__S = 1 - f_S.^2;
end

return
```

MATLAB code - Softmax function

```
function [P_c err_c] = Softmax(W, S, c, opt)

% Initialization
N      = size(S,2);

% Calc. f(S)
f_S = mapS(S, opt);

% Softmax function
exps  = exp(W*f_S);
denom = sum(exps);
for i = 1:opt.C
  P_c(i,:) = exps(i,:) ./ denom;
end

% Extract highest prob.
[temp y] = max(P_c);
y         = y - 1;

if nargout > 1
  % Calc. error rate
  err_c = length(find((c-y) ~= 0)) / N;
end
return
```

MATLAB code - Generation of data

```
function [S, cost , opt] = Generate_S(A, W, c, opt)

% Generate data
% ------------------------------------------------------------------------
% Input parameters
% A                 : Codebook feature matrix
% W                 : Class. weight matrix
% c                 : Target class as vector
% opt.type          : Type of maping function , fs)
%
% Output parameters
% f_S               : Mapped sources
% ------------------------------------------------------------------------


% Initialize s to unform random elements
d        = size(A,2)-1;
N        = length(c);
S        = 1*rand(d+1,N)-0.5;
S_old    = S;
loop     = 1;
cost     = inf;
stop_id = [];

rand('state',sum(100*clock));           % See random generator

while(loop)
  cost_old = cost; it = 0;

  % Map S through non-linear function f(s)
  f_S  = mapS(S, opt);
  f__S = mapS_(S, f_S, opt);            % Derivative of f(s)

  exps  = exp(W*f_S);
  S_    = ( (W'*exps ./ (ones(d+1,1)*sum(exps)) - W(c+1,:)') .* f__S );
  S_    = S_ + opt.alpha*sign(S);

  while (1)
    S_(:,stop_id) = 0;              % Remove gradient for already converged digits
    S = S_old - opt.nuS*S_;         % Take step

    cost = - sum(sum(W(c+1,:)'.*f_S)) + sum(log(sum(exp(W*f_S))));
    cost = cost + opt.alpha*norm(S,1);

    if cost < cost_old
      [y_all err_c] = Softmax(W, S, c, opt);

      % Extract p(c|s) for each digit
      for i = 1:N
        y_c(i) = y_all(c(i)+1,i);
      end
      stop_id = find(y_c > opt.S_p_c_s);        % Save id for conv. digits

      if length(stop_id) == N;                  % Stop if converged
        loop = 0; end
      S_old = S;
      opt.nuS = opt.nuS*opt.accel; break;       % Increase stepsize and exit
    else
      opt.nuS = opt.nuS/2 + eps;
      it = it + 1;
      if it > opt.IT_in                         % Exit if not converged
```

```
        loop = 0;
        S    = S_old;                           % Revert to old S and cost
        cost = cost_old;
        break;
      end
    end
  end
  if opt.plot
    figure(100), showPatch(A*mapS(S,opt), min(10,N), 'single');
  end
end
return
```