CHAPTER ONE

INTRODUCTION

The brain needs to form an internal representation of the outside world to interact with it and does so by representing or coding information in the spatio-temporal activities of neurons. Neurons are complex computational units. They communicate mainly with electrical pulses called spikes, but they also make use of chemical communication. A neuron can be viewed as a temporal binary element; it is either silent or it fires a spike. This is a rather simple view of a neuron (Aur, Connolly and Jog, 2006) but it captures the main means of neural communication (Rieke, Warland, de Ruyter van Steveninck and Bialek, 1996). Figure 1.1 shows the activity pattern of some neurons in the visual cortex; it reveals a pattern in the response of neurons to a repeating stimulus.

1.1 THE NEURAL CODE

The neural code is the “language” of the brain. It contains information about stimuli, it contains the commands the brain sends to muscles, it contains thought etc. Researchers have

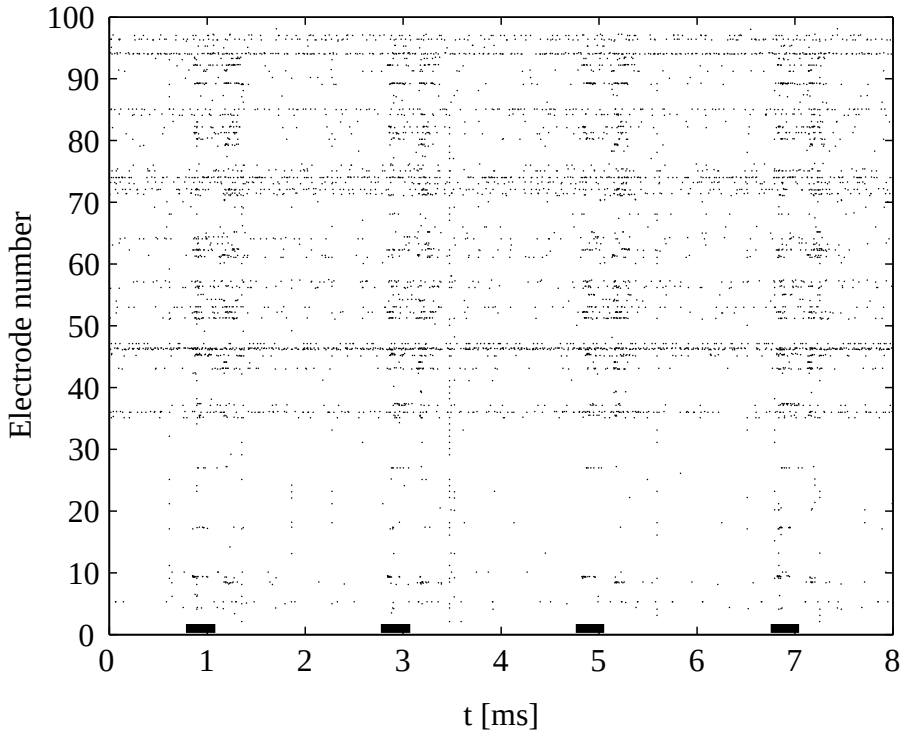
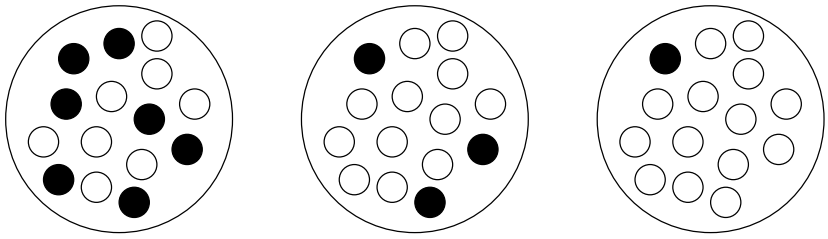


Figure 1.1: Here the activity of 100 neurons in the visual cortex are shown in response to a simple repeating black-white stimulus. Each dot signals a spike. The stimulus is white for 300ms, indicated by the dark bars at the bottom, and black for 700ms. (This data is part of a sample file distributed with the software program CORTIVIS).



(a) Dense coding. (b) Sparse coding. (c) Local coding.

Figure 1.2: In dense coding (a) many of the available neurons are active in the representation of a stimulus, whereas in local coding (c) only one neuron is active in the representation. Sparse coding (b) is a scheme somewhere between dense coding and local coding.

not been able to read the neural code or understand exactly how the brain process information (Bair, 1999; Eggemont, 1998; Theunissen, 2003). This remains an active area of research and experimental work adds to our understanding of the neural code.

It is useful to take a simple view of a neuron in order to compare the neural code with common computational codes. Such a comparison helps us to understand certain aspects of the neural code. Figure 1.2 illustrates some coding schemes for binary elements. Information can be represented by a *dense code* (figure 1.2(a)). For such a code few neurons are required to code many different stimuli but the neurons will be very active. Many of the available neurons will participate in the representation. A particular neuron therefore does not convey that much information about the stimulus, instead the information is distributed among the neurons. A code can also be a *local code* (figure 1.2(c)). In this case there are many neurons available but only one participates in coding a specific stimulus. Both of these extremes are biologically implausible. The local code will require too many neurons to code all the possible stimuli. Even though there are billions of neurons in the brain, there are still too few so that each neuron can code one and only one stimulus. A dense code is also implausible as the neurons will have to be too active.

Generally the lower level sensory neurons are more active than higher level neurons. At the highest level neurons fire on average once every second. The brain adopts a coding scheme somewhere between local coding and dense coding, which is called *sparse coding*(figure 1.2(b)) (Földiák and Young, 1995; Vinje and Gallant, 2000). The optimal level of

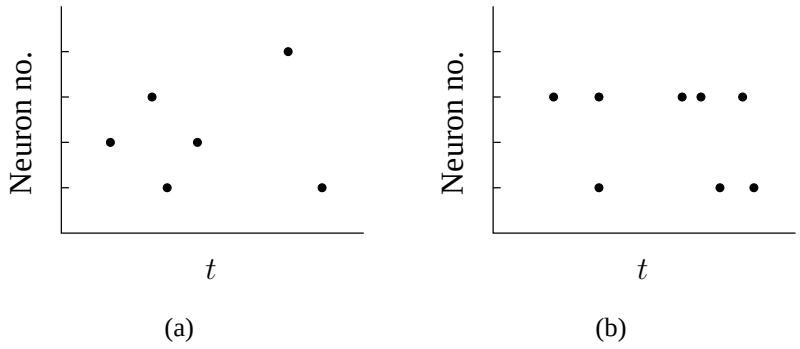


Figure 1.3: The figure shows two spike trains that code the same stimulus. Spike train (b) has a population activity that is more sparse than spike train (a) eventhough it has more spike. This is because population sparseness depends on the number of neurons that participate in a representation and not on the number of spike per representation.

sparseness may be different for various areas in the brain.

1.1.1 SPARSE CODING

Sparse coding is a general principle that is employed in most areas of the brain (Beloozerova, Sirota and Swadlow, 2003; DeWeese, Wehr and Zador, 2003; Laurent, 2002; Vinje and Gallant, 2000; Vinje and Gallant, 2002). There are two forms of sparseness namely, *population sparseness* and *lifetime sparseness*. Population sparseness refers to the fact that only a *small subset* of neurons is active in coding a stimulus, whereas *lifetime sparseness* refers to a particular neuron being seldom active over a long time period (Willmore and Tolhurst, 2001) (see figures 1.3 and 1.4). Both forms of sparseness are present in the brain and they are not identical. For example, each neuron in a group may have a good lifetime sparseness (not very active over long time spans), but the population sparseness will be poor if all the neurons are active at the same time. Whenever we refer to sparseness, it includes lifetime sparseness and population sparseness.

In a sparse code a small subset of neurons participate in coding a particular stimulus. Sparse coding has several advantages over other types of coding. A sparse code can also store more patterns in associative memory than either a dense code (Földiák, 1998; Rolls

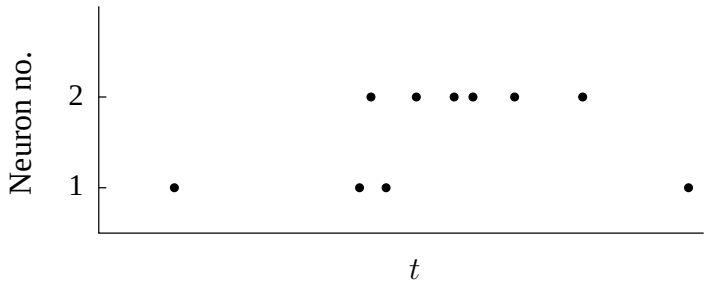


Figure 1.4: The activity of two neurons over a long time period in which several stimuli occured. The lifetime activity of neuron 1 is more sparse than neuron 2 because it fires fewer spikes.

and Treves, 1990) or a local code (1-of-N representation). Sparse coding makes structure in the input more explicit (Olshausen and Field, 2004), it is then easier for other areas in the cortex that receive the sparse code to perform computations on it. Furthermore, evidence suggests that the representation is efficient in an information sense (Vinje and Gallant, 2002), i.e. it contains as much information about the outside world as possible, given the physical constraints of the brain.

1.1.2 EVIDENCE FOR SPARSE CODING

Vinje and Gallant (2000) and Vinje and Gallant (2002) found that neurons in V1 (a large and important visual cortical area) respond sparsely to natural stimuli. Interestingly their responses are more dense for unnatural stimuli. This shows that the neural code is adapted to represent natural stimuli in a sparse manner. Sparse coding has also been observed in the auditory cortex (DeWeese *et al.*, 2003), the olfactory system (Laurent, 2002) and the motor cortex (Beloozerova *et al.*, 2003).

Computational studies have shown that sparse coding leads to receptive fields that resemble the actual receptive fields, again proving that the brain makes use of sparse coding. For most of the studies a *linear generative model* is used. The model linearly adds symbols or dictionary elements together in order to reconstruct a stimulus. The code gives the coefficients with which the dictionary elements have to be scaled before they are added to

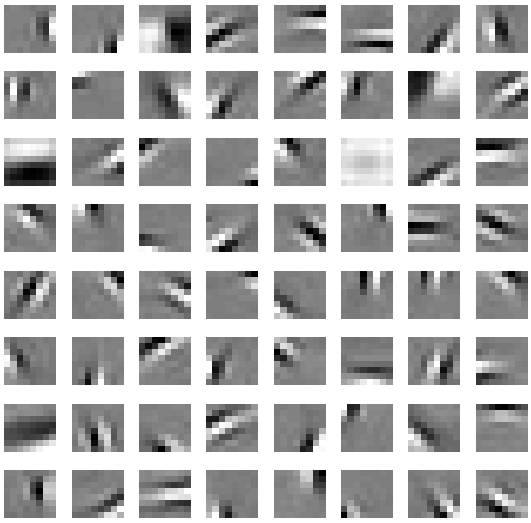


Figure 1.5: The dictionary elements as calculated with SPARSENET (Olshausen and Field, 1996b; Olshausen, 1996). The elements resemble the Gabor filter-like receptive fields of some neurons in V1.

reconstruct the signal. The dictionary is adapted to represent a dataset with sparse codes. Olshausen and Field (1996a) found that for image patches from natural scenes, the adapted dictionary elements are Gabor filter-like receptive fields (figure 1.5). Receptive fields that look similar are found in simple cells of V1. A simple cell responds to an edge that is orientated in the preferred direction of the cell, and is located in a specific area in the receptive field. Simple cells make up about a quarter of the cells in V1 while the remainder of the cells are complex. Complex cells also respond to specifically orientated edges, but the edge can be anywhere in the receptive field. Other complex cells respond to specifically orientated edges that move in a certain direction.

The receptive fields of more complex cells cannot yet be explained by the linear generative model. However a more complex model together with sparse coding shows some correlation between the receptive fields of complex cells and the dictionary elements (Hyvärinen and Hoyer, 2001). Sigman, Cecchi, Gilbert and Magnasco (2001) used a complex model to explain some properties of complex visual cells.

Less work has been done on senses other than sight. Lewicki (2002) found that when a linear generative model codes natural sounds in the time domain, the elements of the trained

dictionary have similar tuning properties to auditory nerve fibres.

1.1.3 OVERCOMPLETE CODES

There are 3500 inner hair cells in the human cochlea. Roughly ten auditory nerve fibres connect to each inner hair cell so that 30 000 nerve fibres extend from the cochlea. Information gathered by the hair cells goes through several stages of processing. The last three stages are the inferior colliculus which has around 392 000 neurons, the medial geniculate body which has 570 000 neurons and the auditory cortex which has about 100 000 000 neurons. There is a great expansion of neurons from the senses to the cortex.

All the sound information about a stimulus is gathered by the hair cells. The amount of information about a stimulus cannot increase from one processing stage to the next. It is however possible that additional information, such as visual clues, may add information about a sound stimulus, but this only appears to happen from the auditory cortex and upwards (Kayser, Petkov, Augath and Logothetis, 2007). Given that the amount of information cannot increase, what advantage is there to having such an expanded representation in higher stages of processing?

Imagine a system similar to the brain that should be able to code an N dimensional *random* signal, but it can use an $N_d > N$ dimensional code. When the dimensionality of the code is more than the dimensionality of the signal the code is *overcomplete*. The code will always contain N active code elements, except if by chance the signal corresponds to a feature that a particular element code for. There is not much advantage in having an overcomplete representation when random signals are coded. However natural stimuli are not random, it has statistical structure. In fact we can only recognize stimuli if it is similar to stimuli we have experienced before.

Consider a code that has as many code elements as the fundamental features that can generate a structured signal. This code can be optimally sparse when each code element corresponds to one of the features. The representation will be sparse as it will have only as many active elements as there are fundamental features in the signal. In this case the

dictionary has to be overcomplete if the number of features is more than the dimensionality of the signal.

The great expansion of the neural representation from lower to higher levels allows neurons to code for very specific features. For example, simple cells in the early visual cortex have Gabor filter-like receptive fields - their responses are broadly tuned. Higher up, in the primary visual cortex neurons respond to more complex features, such as edges, lines, or gratings. In even higher visual areas like V4, neurons respond to more complex features such as faces and hands (Földiák, 1998). There even exist "grandmother" cells (Quirago, Reddy, Kreiman, Koch and Fried, 2005). These neurons are so finely tuned that they only respond to very specific stimuli, such as the thought or sight of the proverbial grandmother. This shows that some neurons in the higher levels of the brain use the one extreme form of sparse coding, namely local coding.

1.2 SPARSE CODING FOR PATTERN RECOGNITION

Sparse overcomplete coding should perform well as a means to pattern recognition because it can capture the underlying structure in the signal. It is a powerful method to extract statistically significant features from a signal.

Most of the published work on sparse coding focuses on the properties of the adapted dictionary, algorithms that can find sparse codes or on models that are more complex than the linear generative model. There are some studies that have used sparse coding to create a feature set for sound or speech recognition tasks. The studies follow the same general approach: first a sound signal is encoded into a spike train, then the recognition task is performed by decoding the spike train. Cho and Choi (2005) perform sound classification with spikes. They classify a sound as belonging to one of ten classes, which include male speech, foot steps and flute sounds. Näger, Storck and Deco (2002) show that transitions between vowels can be classified by learning the delays between spikes. Kwon and Lee (2004) use independent component analysis (ICA) to extract features from speech in order to do phoneme recognition. ICA is an algorithm that provides a sparse representation of a

signal. Some studies illustrate isolated digit recognition (Loiselle, Rouat, Pressnitzer and Thorpe, 2005; Mercier and Séguier, 2002; Verstraeten, Schrauwen, Stroobandt and Campenhout, 2005) while Holmberg, Gelbart, Ramacher and Hemmert (2005) demonstrate isolated letter recognition. All of these studies consider only isolated samples.

Very few studies have used sparse coding on practical image recognition problems. Sun, Zhou, Ma and Wang (2001) applied sparse coding to face recognition. Sparse coding achieved a recognition rate of 95% compared to 89% for Fisherface (a popular face recognition method).

All of the work mentioned above use inputs that are already segmented. In other words, it is not necessary to first divide the input into segments before it can be classified. However in this study we use sparse coding to do *continuous speech recognition*. The classification algorithm should be able to segment the signal.

The aims of the study are discussed in the last section of this chapter, but first we give an overview of the speech recognition problem and briefly discuss current speech recognition systems.

1.3 SPEECH RECOGNITION IN GENERAL

Automatic speech recognition (ASR) is a convenient interface between man and machine, because speech is such a natural way for people to convey information. It can also allow people that are unfamiliar with some technologies or some disabled people to make use of modern technology.

An ASR system can be categorized as speaker-dependent or speaker-independent; as an isolated word recognizer or a continuous speech recognizer; as working with a limited vocabulary or a large vocabulary. The most versatile ASR task would be to do speaker-independent, continuous speech recognition with a large vocabulary. This is also the most difficult ASR task, and the performance of computers on this task does not yet come close to that of humans. But other types of ASR systems have successfully been applied to everyday

ASR problems.

Persons with certain disabilities, such as quadriplegics, may find limited vocabulary, isolated word recognition systems very useful. With such a system they could give commands to a television set, a motorized bed etc. Some cars also use a similar system to allow the driver access to functions that are usually situated on the steering wheel, to operate the navigation system or the car stereo.

Speaker-independent continuous speech recognition with limited vocabulary is often used in switchboard applications, where a caller gives the name of the person he would like to speak to, and his call is automatically forwarded to that person. Other systems allow a caller to book movies, make hotel or flight reservations.

Dictation systems employ state-of-the-art technology continuous speech recognition. These systems can produce documents much faster than many people can type, once it is trained on a specific user, and if the user learns to speak a bit slower and clearer. A dictation software program is distributed as a speaker-independent system, but the accuracy of an off-the-shelf system is not good enough for the system to be useful. The user has to train the system for one or two hours before the recognition accuracy increases to a useful level. After the system has been trained that much, it is not really speaker-independent anymore.

Unfortunately the performance of a truly speaker-independent ASR system, are not yet good enough to be used in everyday applications. These systems are very susceptible to noise, even office background noises can cause their performance to degrade considerably.

There is therefore a need for ASR to move closer to human-like speech recognition.

1.4 CURRENT SPEECH RECOGNITION SYSTEMS

The most common ASR systems use across-frequency features to recognize speech. A signal is divided into small, usually 20ms, segments and some type of frequency analysis of each segment yields the feature set. The most widely used feature set is the Mel Frequency Cepstrum Coefficients (MFCC) and time derivatives thereof (Rabiner and Juang, 1993).

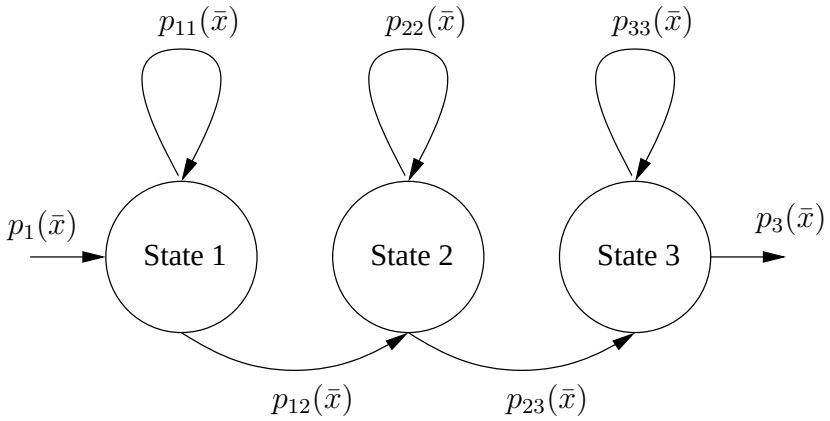


Figure 1.6: An acoustic unit is modeled with a three state Hidden Markov model. The probability of transition from one state to the next is a function of the feature vector $\bar{x}$.

The features serve as inputs to models of acoustic units, which can be words, phonemes, triphones etc. An acoustic unit model is usually a 3 or 5 state Hidden Markov Model (HMM) (see figure 1.6). The probability of transition from one state to another is determined by the feature vector for a particular segment. By concatenating the unit models, bigger units can be modelled, and eventually sentences. A language model ensures that the acoustic units are concatenated in a language consistent manner. A very basic language model would give the probability of a specific acoustic unit following another specific acoustic unit.

Continuous ASR aims to recognize sentences given a sequence of feature vectors. It is possible to find the most likely sequence of acoustic units, by finding the sequence of models that best fit the feature vectors and the language model. Once the sequence of acoustic units is determined, a post processor combines the units into sentences.

This approach to ASR does not contain much prior knowledge of speech, as information on the human communication process is limited. It rather makes use of a large amounts of data to self discover relevant patterns in speech within the framework of an HMM. This approach should work very well, provided there is enough information and provided the HMM framework can model speech decoding adequately. Later in this chapter we will show that there is an alternative speech decoding model that fit speech data better than the HMM approach.

1.4.1 SOME PROBLEMS WITH CURRENT SYSTEMS

There are a few reasons why the performance of current systems does not compare to that of humans. These reasons relate to the feature set, the acoustic units and the HMM framework.

Temporal dynamics play an important role in human speech recognition. It has been found that the relevant information for speech recognition lies in the modulation range of 1 to 15Hz (Hermansky, 1998). That is because the vocal tract cannot change faster than 15Hz, and information outside this range can therefore not contain anything relevant to speech recognition. Perceptual experiments (Allen, 1994; Kral, 2000; Moller, 1999; Shannon, Zeng, Kamath, Wygonski and Ekelid, 1995) have further showed that temporal information in speech is much more important than spectral information.

Current systems use an across-frequency feature set where each segment is assumed to be stationary and independent of neighbouring segments. This assumption does not take the temporal dynamics of speech into account at all. The problem is solved in part by augmenting the basic feature set with its time derivatives. Temporal dynamics are modelled with the HMM.

Across-frequency features in a short time segment is susceptible to noise, because it is difficult to separate noise from signal over a short time interval. This is one of the main reasons why current ASR systems fail in noisy environments, even when humans do not find it a difficult task.

Another problem with current ASR systems lies with the acoustic units. These units are very basic, and smaller units have been used more successfully that larger units. This is in contrast to human perception of speech, where the basic unit is probably on the syllable level (Nguyen and Hawkins, 2003).

A further shortcoming of current ASR systems is in the way the most likely sequence is found. Longer sounds will carry more weight than shorter sounds, because there is a cost associated with each time segment. The most likely sequence may end up not to be the best choice.

Finally the HMM framework may not be well suited to model speech. This notion is supported by the fact that huge amounts of data is required to train such a system, which even after training, performs poorly in conditions that are not very close to that under which the training set was gathered.

It's reasonable to assume that the best speech decoder is the auditory cortex. We should therefore look to include aspects of the cortex in ASR systems in order to improve them.

1.5 SPARSE CODING FOR SPEECH RECOGNITION

Psychological studies have showed that mammalian auditory cortical neurons respond to spectro-temporal sound patterns (deCharms, Blake and Merzenich, 1998; Shamma, 2001), that is patterns across-frequency *and* across-time.

There are some models that use across-time processing (Hermansky and Sharma, 1999; Hermansky and Morgan, 1994; Ikbal, Magimai.-Doss, Misra and Bourlard, 2004), and others that use spectro-temporal features (Kleinschmidt, 2002; Klein, König and Körding, 2003; Kwon and Lee, 2004). These models do not achieve the same recognition results as state-of-the-art ASR systems in relative noise free environments, but they have performed better than HMMs in noisy environments.

Sparse coding with an overcomplete dictionary can give a code whose features are meaningful spectro-temporal sound patterns. The linear generative model (LGM) easily fit into a sparse coding scheme. The model reconstructs a signal of a preset length using a dictionary

$$\bar{x} = \mathbf{\Phi}\bar{a} \tag{1.1}$$

The signal $\bar{x}$ is encoded by a set of coefficients $\bar{a}$, one for each dictionary element in $\mathbf{\Phi}$ (the columns in $\mathbf{\Phi}$ contain the dictionary elements).

Principle Component Analysis (PCA) can be seen as a generative model. PCA chooses a dictionary such that the elements are all orthogonal, and such that the elements point in the directions ordered from the largest to the lowest variance of a given set of signals. Another choice of dictionary is given by Independent Component Analysis (ICA). ICA tries to find

dictionary elements that will result in a code whose coefficients are independent of each other for a given set of signals. Both PCA and ICA use a complete dictionary, i.e. for an N dimensional signal vector, $\mathbf{\Phi}$ is an $N \times N$ sized matrix. As already mentioned there are various advantages to having an overcomplete dictionary, where $\mathbf{\Phi}$ is an $N \times N_d$ sized matrix, and $N_d > N$.

For an overcomplete dictionary the code cannot be uniquely determined. It is then possible to search for a sparse code from all the possible codes. The sparse code of a LGM is a pulse like structure, similar to the spike structure by which biological neurons communicate. Patterns in this structure can be related to specific events in the signal, such as spoken words. Another benefit of encoding a signal with a pulse structure is that the decoding algorithms are not such a strong function of time as the decoding algorithm in current ASR systems is.

1.5.1 AIMS OF THIS STUDY

The *primary aim* of this study is to use overcomplete sparse coding for pattern recognition, in particular continuous speech recognition. This study will serve as an initial investigation into sparse coding for real-world pattern recognition problems. It will show what aspects of the implementation of sparse coding are important and also what the shortcomings of current methods are. It will further highlight the factors that limit the use of sparse coding for pattern recognition.

We use the LGM to select codes that can represent a signal. The dimensionality of the code is very high, especially as the code is overcomplete. The current algorithms that find sparse codes of such high dimensional problems are either computationally expensive or they yield a code that is not well suited for pattern recognition. We therefore need to develop an algorithm that satisfies both requirements.

A *secondary aim* is to see what improvements sparse coding can bring to current speech recognition systems. Overall sparse coding via a generative model seems to be a better model for speech recognition than MFCC coefficients and HMMs, because:

- it uses spectro-temporal features,
- it does not prescribe which acoustic units should be used, but the dictionary is free to choose any unit,
- it combines the basic units into sentences by using an algorithm that is not a linear function of time, and
- the properties of the model correlate well with that of the auditory cortex, making it biologically more plausible.

This study will show whether these benefits actually improve speech recognition performance, and whether there is merit in doing further investigation along these lines.

In chapter 2 we show how a signal can be represented by a sparse code. The sparse code resembles a spike train (the spatio-temporal activities of neurons). Chapter 3 discuss our method for decoding the sparse code; the chapter shows how to determine the words that are coded by the spike train.

CHAPTER TWO

SPARSE CODING

As mentioned earlier, we would like to use the idea of sparse coding in a speech recognition system. This requires that a stimulus or signal be transformed into a sparse code, and that patterns in the sparse code be associated with spoken words. In this chapter we will first look at ways to find a sparse code that will represent a signal. We then look at the implementation issues that arise when sparse coding is applied to speech recognition. Finally, we give the results of sparse coding for speech. The next chapter will discuss the pattern recognition problem.

2.1 THE LINEAR GENERATIVE MODEL

Signals are often represented as a linear sum of elementary signals; for example, with the Fourier transform a signal is represented as a linear combination of sinusoids. Such a representation may have favourable properties. It may, for example, highlight important elements

or objects that are present in the signal, or it may distinguish between components that behave in characteristic fashion.

We want to transform a speech signal into a sparse representation. Thus far the most successful approach is to use a linear generative model (LGM) (Lewicki and Sejnowksi, 1999; Lewicki, 2002):

$$\overline{x} = \mathbf{\Phi}\overline{a} + \overline{\epsilon} \tag{2.1}$$

generally, $\overline{x}$ is the N dimensional signal, $\overline{a}$ the N_d dimensional representation or code, $\mathbf{\Phi}$ the $N \times N_d$ transformation matrix and $\overline{\epsilon}$ is an N dimensional additive noise term. $\bar{x}_R = \mathbf{\Phi}\overline{a}$ is the reconstructed signal. The elements a_i in the code reveal which objects (the columns in $\mathbf{\Phi}$) are present in the signal.

A representation of the above form is only helpful if the dictionary elements capture in some way the structure of the signal. A random dictionary would not be nearly as useful as a dictionary of sinusoids if one is looking for the frequency content of a signal.

The type of model in equation 2.1 is also successfully applied in blind source separation (Lee, Lewicki, Girlami and Sejnowski, 1999), in image denoising, image compression (Lewicki and Olshausen, 1999; Lewicki and Sejnowski, 2000; Kreutz-Delgado, Murray, Rao, Engan, Lee and Sejnowksi, 2003) and digital water marking (Bounkong, Toch, Saad and Lowe, 2003). The transform $\mathbf{\Phi}$ is named differently depending on the area of application. It can be called a basis matrix, mixing matrix, or dictionary. The columns in $\mathbf{\Phi}$ are called synthesis functions, basis functions, words or dictionary elements. We will use the term dictionary to refer to $\mathbf{\Phi}$ and dictionary elements to the refer to the columns in $\mathbf{\Phi}$.

As we are interested in speech recognition, we will use the generative model to do feature extraction. There is a lot of structure in speech, as in all natural stimuli. If the dictionary elements reflect this structure, then the code will reveal the basic elements that make up a particular signal.

The dictionary determines the features that will be extracted and also the properties of the code. Depending on the application, the best choice of a dictionary may not be a predetermined set such as sinusoids or wavelets, but may be data dependent. For example, in

image compression, Fourier bases which are data independent achieve a 5.34 bits/pixel coding efficiency, while a data dependent dictionary such as ICA achieves a significantly better coding efficiency of 4.71 bits/pixel (Lewicki and Olshausen, 1999).

If the number of dictionary elements equals the signal dimension ($N = N_d$), we speak of a *complete* dictionary. An *overcomplete* code ($N_d > N$) has several advantages over a complete code. It is more stable in the presence of signal noise (Kreutz-Delgado *et al.*, 2003), small changed in the signal does not cause great disturbances in the code; it can give a sparse representation of the signal; and the code can be more efficient in an information sense (Kreutz-Delgado *et al.*, 2003; Lewicki and Olshausen, 1999).

However if the dictionary is overcomplete, then the code may not be uniquely determined, because there are many different codes that can represent the same signal. In this case a sparseness function can be used to choose the best code $\bar{a}^*$ from among the viable codes. The fact that the code cannot uniquely be determined from the signal signifies that the code is not linearly dependent on the signal. The sparse code for $\bar{x}_3 = \bar{x}_1 + \bar{x}_2$ does not have to be simply the sum of the sparse codes for $\bar{x}_1$ and $\bar{x}_2$.

Section 2.4 will look at the code selection problem in more detail. It can generally be stated as an optimization problem that is a function of the reconstruction error $R(\bar{a}) = \|\overline{x} - \mathbf{\Phi}\overline{a}\|^2$ and the sparseness function $S(\bar{a})$ (S is small for sparse codes). For example the code $\overline{a}^*$ to represent a signal $\overline{x}$ is

$$\overline{a}^* = \min_{\overline{a}} S(\overline{a}) \quad \text{such that} \quad R(\bar{a}) \leq \epsilon \tag{2.2}$$

with ϵ the noise level. Another way of formulating the optimization problem is

$$\overline{a}^* = \min_{\overline{a}} R(\bar{a}) + \lambda S(\bar{a}) \tag{2.3}$$

where λ sets the balance between reconstruction and sparseness.

The sparseness function plays an important role in the code that will be selected to represent a signal. In section 2.3 we turn our attention to the sparseness function S.

2.1.1 NEURAL REPRESENTATION AND THE CODE

A nonzero code element corresponds to a spike in the neural code. There is a discrepancy between the neural code and the code of a LGM. A neural spike is a binary event whereas a code element can take on any real value. We can interpret positive real values as some number that is related to the number of neurons that fire spikes simultaneously. The brain appears to be redundant in this manner: more than one neuron can code the same information, which helps the brain to deal with physical damage. Negative real values are more difficult to interpret (although they can be viewed as inhibitory spikes). We choose to rather constrain the code elements to be nonnegative $a_i \geq 0$.

It has not yet been shown that a nonnegative sparse code is in some way better than a sparse code that includes negative terms, but it is easier to interpret physiologically, and, as we will show later, it provides a lower bound to a possible nonconvex minimization problem.

2.2 SPARSE CODING OF LETTERS

Here we give a simple application of sparse coding of letters to illustrate the LGM and sparse codes. The dataset appears in figure 2.1. It has 26 samples, one for each letter of the alphabet. Noise can be added to the samples to create a more realistic dataset, but for the purpose of illustration we consider only the noiseless case.

The dimensionality of the dataset is the number of pixels per letter, which is 15. We use an overcomplete dictionary of 20 elements to reconstruct the letters. The code will be maximally sparse if the dictionary has at least as many elements as there are letters. The elements of the optimal dictionary will then be exactly the dataset. However an overcomplete code will still be sparse even if it is not maximally sparse.

The dictionary is adapted to the data and appears figure 2.2. It contains dictionary elements that fit some letters almost exactly, such as elements 1, 5, 11 and 12; some elements are very similar to letters of the alphabet such as 2, 7, 8 and others; element 10 is not like any letter in the dataset. Inspection of the results reveals that element 10 is used, with very

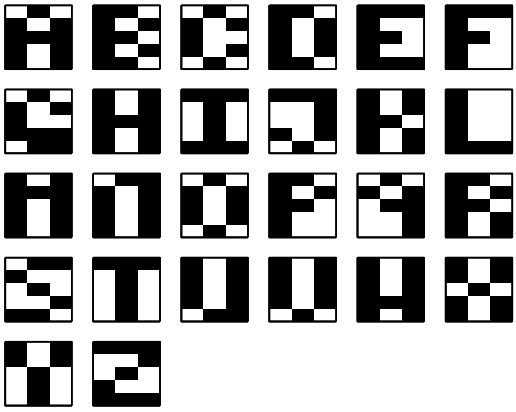


Figure 2.1: The letters that make up the dataset we use to illustrate sparse coding.

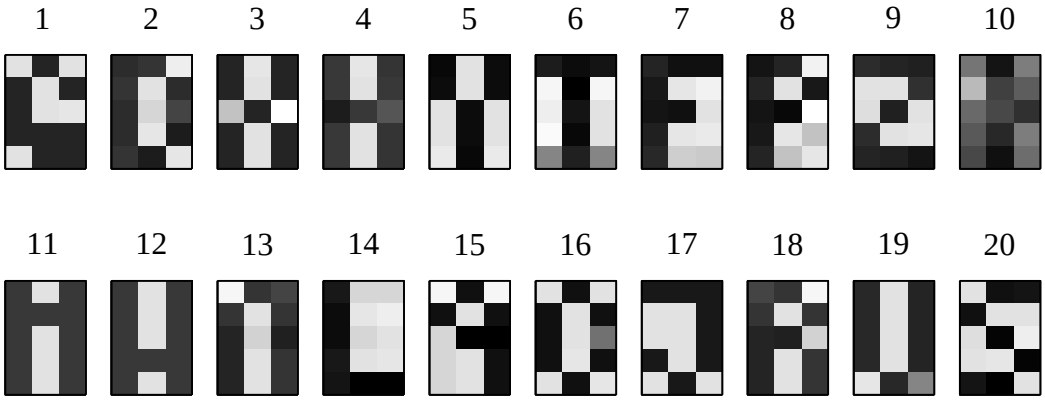
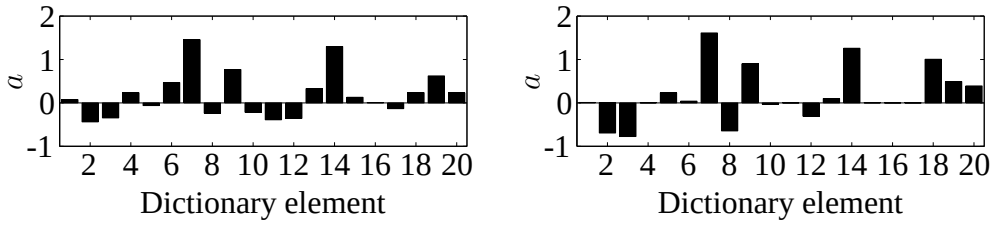


Figure 2.2: The overcomplete dictionary that is adapted to the data. Some dictionary elements have negative components (white represents a value of -0.05 and black represents 0.4).

small code values, in coding the letters “A” and “G”.

Figure 2.3 shows two different codes based on the dictionary in figure 2.2, that both reconstruct the letter “E” perfectly. The reconstruction is simply a sum of scaled dictionary elements, where the scaling factor for each element is given by the code. The codes make significant use of dictionary elements 7 and 14. These two elements are similar to the letters “F” and “L”; their combination in turn is similar to the letter “E”. The code in figure 2.3(b) is sparser than the code in figure 2.3(a) as it has fewer nonzero code elements. This illustrates the point that for an overcomplete dictionary there are many codes to represent a signal. In



(a) A dense code. (b) A sparse code.

Figure 2.3: The figure shows two codes that perfectly reconstruct the letter "E". However, the code in (b) is sparser than the code in (a) because it has fewer nonzero code elements.

such a case a sparseness function can be used to select a sparse code.

2.3 MEASURES OF SPARSENESS

The neural code is sparse, it has both population sparseness and lifetime sparseness. We would like to measure and quantify sparseness, so that a single code can be selected to represent a signal with the LGM.

The neural code is also efficient in an information sense. This implies that the activities of two neurons cannot be correlated, otherwise there will be reduncies in the code. It follows then that the activities of any two neurons should be *independent*.

2.3.1 POPULATION SPARSENESS

At first we will address the measurement of population sparseness. Let the population sparseness function be

$$S_{pop}(n) = S(\overline{a}^{(n)}) \tag{2.4}$$

with $\overline{a}^{(n)}$ the code for stimulus n.

The values of the code elements have to be independent which implies that the sparseness function should be permutation invariant i.e. the sparseness value should not be dependent on the order of the individual components. A sufficient condition for a function to be permu-

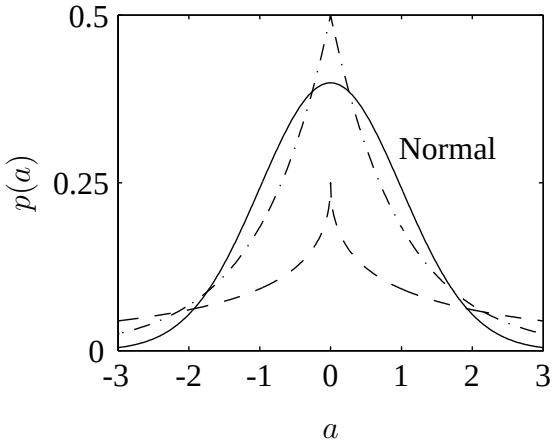


Figure 2.4: Code element values that follow a normal distribution are not considered sparse. Generally a code will be sparse if the probability that a code element value is close to zero is high. The chain ($p(a) = 0.5\exp(-|a|)$) and dashed ($p(a) = 0.25\exp(-|a|^{0.5})$) lines display probability distributions of code element values that are sparse.

tation invariant is separability

$$S(\overline{a}^{(n)}) = \sum_i S(a_i^{(n)}) \tag{2.5}$$

Suppose a neuron is represented by a code element a_i that only take on a binary value, then an appropriate sparseness function would be the l_0 norm, $S(\overline{a}) = \sum_i |a_i|^0$ where $0^0 = 0$. This norm simply counts the number of active or nonzero elements used to represent a stimulus. The l_0 norm will be small for if the code for that stimulus is sparse.

Sometimes a neuron is represented by a real-valued code element a_i, such as a perceptron or a code element in a LGM. Now the l_0 norm may not be appropriate anymore. A sparseness function for a real-valued code may not only consider the fact that a code element is nonzero, but also the exact value of that code element. For real-valued codes the probability distribution $p(\overline{a})$ of the values that code elements assume can indicate the sparseness of that code. The code will be sparse when the probability that an element value is zero or small is much larger than the probability that an element value is large. Section 2.3.3 lists the requirements for a sparseness function to ensure a sparse code. Figure 2.4 shows as an illustration a few probability distributions for which only the normal distribution is not consistent with a sparse code.

Interestingly in literature the l_0 norm is sometimes assumed to be the best measure for sparseness even though the code may be real valued (Field, 1994; Kreutz-Delgado *et al.*, 2003; Wipf and Rao, 2006).

We will show in section 2.4.2 how a sparseness function for a real-valued code can be derived from a probability distribution of code element values.

2.3.2 LIFETIME SPARSENESS

A neuron that is seldom active in the representation of a dataset has high lifetime sparseness. The lifetime sparseness function should not be dependent on the order in which stimuli are coded, which implies that the lifetime sparseness function is a permutation invariant function. For the sake of simplicity let the lifetime sparseness function and the population sparseness function use the same sparseness function. Now the lifetime sparseness function of code element i for all stimuli in the dataset is

$$S_{life}(i) = \sum_n S(a_i^{(n)}) \tag{2.6}$$

In this case the total population sparseness function for all stimuli in a dataset equals the total lifetime sparseness function of all code elements over the same dataset.

$$\sum_n S_{pop}(n) = \sum_n \left[\sum_i S(a_i^{(n)}) \right] = \sum_i \left[\sum_n S(a_i^{(n)}) \right] = \sum_i S_{life}(i) \tag{2.7}$$

There is an important property that arises when lifetime sparseness and population sparseness use the same sparseness function: by selecting the most sparse population code for a *single* given stimulus, we are also selecting the code that will yield the greatest lifetime sparseness over the *entire* dataset.

2.3.3 HOW TO ENSURE SPARSENESS

The sparseness function should have certain properties for it to produce a sparse code. It is generally believed that when the probability distribution of a code element value is peaked at

zero and is heavy-tailed, then that code element is consistent with a sparse code (Olshausen and Field, 1997; Lewicki and Olshausen, 1999; Olshausen and Field, 2004). Probability distributions in the form of

$$p(\bar{a}) \propto \sum_i e^{-S(a_i)} \tag{2.8}$$

are often used to describe the distribution of sparse codes. Several sparseness functions have been proposed (Olshausen and Field, 1997; Lewicki and Olshausen, 1999; Lewicki, 2002): $S(a) = \beta \ln(1 + a^2)$, $S(a) = -\exp^{-a^2}$, $S(a) = |a|^q$ with $0 < q < 2$, and others.

Kreutz-Delgado *et al.* (2003) have derived sufficient conditions for functions to be valid measures of sparseness, i.e. functions that promote sparseness. The conditions are:

- The function should be permutation invariant. A function is permutation invariant if its value is independent of the order of its components.
- $S(|a| + \Delta a) < S(|a|) + S(\Delta a)$, i.e. the cost of increasing the amplitude of an existing spike by $\Delta a > 0$ should be less than the cost of adding a new spike with that same amplitude to the code.

This implies that $S(a) = |a|^q$ is a valid sparseness functions only for $0 < q \leq 1$. The second condition is not satisfied for $1 < q < 2$, therefore the general belief that all supergaussian functions ($0 < q < 2$) enforce sparseness is not well-grounded.

We use the sparseness function

$$S(a) = a - \frac{\beta}{2} a^2 \tag{2.9}$$

within the bounds $0 \leq a \leq 1/\beta$. The lower bound ensures that components are positive while the upper bound is needed to satisfy the second condition above. Figure 2.5 gives a contour plot of equation 2.9 for a two component code. It shows that the sparseness function is lower for points closer to the axes, i.e. sparser codes.

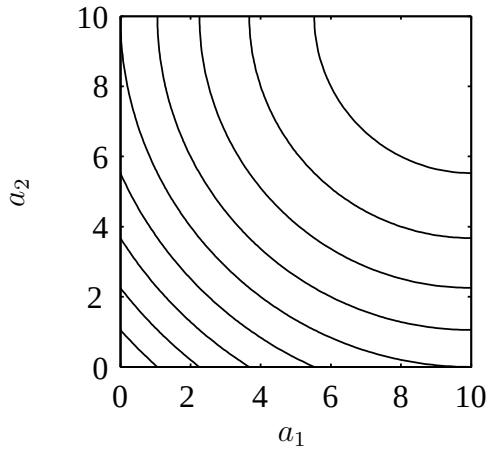


Figure 2.5: A countour plot of $S(\bar{a}) = \sum_i a_i - \frac{0.1}{2}{a_i}^2$, $S([0,0]) = 0$ and $S([10,10]) = 10$. It shows that the sparseness function is smaller for sparse codes. For example, $S([5,0]) = 3.75$ while $S([2.5,2.5]) = 4.375$.

2.4 FINDING A SPARSE CODE

A sparse code should be one that represents a signal well, but that is also sparse. Sparse codes are usually found in one of two ways: by viewing the LGM in a probabilistic framework and finding the most likely code, or by explicitly stating the problem as a mathematical optimization problem. The following sections expand on both approaches.

2.4.1 EXPLICITLY STATING THE MATHEMATICAL OPTIMIZATION PROBLEM

The sparse code $\overline{a}^*$ that represents a signal is given in equation 2.2

$$\overline{a}^* = \min_{\overline{a}} S(\overline{a}) \quad \text{such that} \quad R(\bar{a}) \leq \epsilon$$

where the reconstruction error is $R(\bar{a}) = \|\overline{x} - \mathbf{\Phi}\overline{a}\|^2$. It is useful to state the sparse coding problem in this way when the sparseness function is nondifferentiable.

Consider a sparse coding problem based on the l_0-norm. The problem of minimizing l_0 while ensuring that reconstruction error is small enough is NP-hard and is a combinatorial

problem. A class of algorithms that attempt to solve this problem is termed *basis selection*. Of the many basis selection algorithms, *matching pursuit* (MP) is one of the most basic.

According to (Mallet and Zhang, 1993) matching pursuit

> ... decomposes any signal into a linear expansion of waveforms that are selected from a redundant dictionary of functions. These waveforms are chosen in order to best match the signal structures.

MP is an iterative algorithm that adds one element at a time to the code, until the reconstruction error falls below a predetermined threshold. The element that is added to the code during the kth iteration is

$$a_i^{(k)} = \max_i \frac{(\bar{x} - \bar{x}_R^{(k-1)})^T \cdot \Phi_i}{\|\Phi_i\|^2} \tag{2.10}$$

$\bar{x}_R^{(k-1)} = \mathbf{\Phi}\bar{a}^{(k-1)}$ is the reconstruction at iteration $k-1$. MP therefore adds during each iteration that single element to code that will most reduce the norm of the residue $\left\|\bar{x} - \bar{x}_R^{(k-1)}\right\|$. Even though MP does not explicitly consider sparseness, it still yields a sparse solution because the algorithm is stopped before it can perfectly reconstruct the signal.

If we view $R(\bar{a}) = \|\overline{x} - \mathbf{\Phi}\overline{a}\|^2$ as a cost function that needs to be minimized, then MP minimizes the cost function by moving the solution along one dimension at a time.

MP is generally faster than *gradient based* algorithms (Perrinet, 2004a). However we show in section 2.7.2.1 that codes from pure MP are not suitable for pattern recognition problems, as small changes in the signal cause big changes in the code.

2.4.2 A PROBABILISTIC APPROACH

We have mentioned that the sparseness of real-valued code elements can conveniently be measured according to the probability distribution of the code element values $p(\bar{a})$. In this section we view the LGM in a probabilistic framework in order to find a sparse code.

For the case of no noise ($\epsilon = 0$) and with a Laplacian prior for the code elements, there are methods that can find the code efficiently by maximizing sparseness while constraining

the solution so that $\bar{x} = \mathbf{\Phi}\bar{a}$ (Chen, Donoho and Saunders, 1998). Approaches to finding the code of a model that includes nonzero noise ($\epsilon > 0$), are independent factor analysis (Attias, 1999), an expectation maximization algorithm (Girolami, 2001) and the FOCUSS algorithm (Kreutz-Delgado *et al.*, 2003). These approaches can only solve problems that can be cast into the form of equation 2.1. More general approaches that also apply to other forms of equation 2.1 (for example time-dependent generative models) are gradient based algorithms (Olshausen, 2002).

Barlow's hypothesis states that we should be looking for the most efficient codes to represent stimuli. An efficient code $\bar{a}$ is one that requires few bits to encode the source output $\bar{x}$. According to Shannon's source coding theorem, a source cannot be coded with fewer bits than its entropy H

$$L \geq H(X) = \int p_{true}(\bar{x}) \ln \frac{1}{p_{true}(\bar{x})} d\bar{x} \tag{2.11}$$

L is the average number of bits needed to encode the source, it is termed the average code length. $p_{true}(\bar{x})$ is the actual probability distribution of the data. We do not know the actual probability distribution, instead it is modelled with a distribution $p(\bar{x})$, where some parameters of the distribution can be changed to better fit the data. The more $p(\bar{x})$ deviates from $p_{true}(\bar{x})$, the less efficient the code becomes

$$L \geq \int p_{true}(\bar{x}) \ln \frac{1}{p(\bar{x})} d\bar{x} \tag{2.12}$$

$$\geq \int p_{true}(\bar{x}) \ln \frac{1}{p_{true}(\bar{x})} d\bar{x} + \int p_{true}(\bar{x}) \ln \frac{p_{true}(\bar{x})}{p(\bar{x})} d\bar{x} \tag{2.13}$$

The second term on the right-hand side is the *Kullback-Leibler divergence* (*KL*). It measures how closely the probability distribution of signals from a model approximates the true distribution. Codes that minimize this term will be efficient.

Minimizing *KL* corresponds to maximizing $\langle \ln p(\bar{x}) \rangle$ since

$$\langle \ln p(\bar{x}) \rangle = \int p_{true}(\bar{x}) \ln p(\bar{x}) d\bar{x} \tag{2.14}$$

differs from *KL* by a quantity that does not depend on $p(\bar{x})$. $p(\bar{x})$ is found by marginalizing over $\bar{a}$:

$$p(\bar{x}) = \int p(\bar{x} \mid \bar{a}) p(\bar{a}) d\bar{a} \tag{2.15}$$

We need to specify the probability distributions of $p(\bar{x} \mid \bar{a})$ and $p(\bar{a})$. If we assume additive normally distributed noise on the signals, then the noise term in

$$\bar{x} = \mathbf{\Phi}\bar{a} + \bar{\epsilon} \tag{2.16}$$

is a random vector with entries that are sampled from $\mathcal{N}(0, \sigma_n^2)$. Now it is possible to specify:

$$p(\bar{x} \mid \bar{a}) \propto \exp(-\frac{\|\bar{x} - \mathbf{\Phi}\bar{a}\|^2}{2\sigma_n^2}) \tag{2.17}$$

The choice of $p(\bar{a})$ is limited by the fact that $\bar{a}$ needs to be sparse. We assume that the components of $\bar{a}$ are independent and that they follow the given distribution:

$$p(\bar{a}) = \prod_i p(a_i) \propto \prod_i \exp(-S(a_i)) \tag{2.18}$$

$S(a_i)$ should be a suitable sparseness function.

$p(\bar{x})$ is found by integrating over all possible $\bar{a}$ (see equation 2.15). It is generally intractable to evaluate this integral exactly. If there is no noise and if the dictionary is complete, the integral can be evaluated, which then leads to the ICA algorithm (Olshausen and Field, 1997; Cardoso, 1997). There are however several ways to approximate the integral. One is to sample a number of points from the posterior $p(\bar{a} \mid \bar{x}) \propto p(\bar{x} \mid \bar{a})p(\bar{a})$, and to approximate the integral by a sum at these points. Another is to approximate the terms inside the integral with a normal distribution centred on the maximum posterior mode $\bar{a}^*$ (Lewicki and Olshausen, 1999; Lewicki and Sejnowski, 2000):

$$\bar{a}^* = \max_{\bar{a}} p(\bar{a} \mid \bar{x}) \tag{2.19}$$

after which the integral in equation 2.15 becomes analytically solvable.

Alternatively, the integral can be approximated by just sampling at the maximum posterior mode (Olshausen and Field, 1997). This last approximation neglects the volume under the integral, and assumes that the term in the integral is a delta function. We will use this approximation for its simplicity although it has two significant drawbacks. Firstly, the actual problem we solve is only an approximation to the true problem we would like to solve. Secondly, it is shown in section 2.5 that the optimal dictionary will be one for which the dictionary elements have norms that approach infinity.

The optimization problem in equation 2.19 can be simplified. By using Bayes' theorem together with the fact that $p(\bar{x})$ is not a function of $\bar{a}$ we have

$$\begin{aligned} \bar{a}^* &= \max_{\bar{a}} \ln p(\bar{a} \mid \bar{x}) && (2.20) \\ &= \max_{\bar{a}} \ln \frac{p(\bar{x} \mid \bar{a})p(\bar{a})}{p(\bar{x})} && (2.21) \\ &= \max_{\bar{a}} \ln p(\bar{x} \mid \bar{a})p(\bar{a}) && (2.22) \end{aligned}$$

Maximizing $p(\bar{a} \mid \bar{x})$ is equivalent to maximizing $\ln p(\bar{a} \mid \bar{x})$ since the logarithm is a monotonic function.

The sparse code $\bar{a}^*$ is the solution to the minimization problem

$$E(\bar{a} \mid \bar{x}) = \frac{1}{2\sigma_n^2}\|\bar{x} - \mathbf{\Phi}\bar{a}\|^2 + \sum_i g(|a_i|) \tag{2.23}$$

where we have made use of equations 2.17 and 2.18. E can also be written as

$$E(\bar{a} \mid \bar{x}) = \|\bar{x} - \mathbf{\Phi}\bar{a}\|^2 + \lambda S(\bar{a}) \tag{2.24}$$

with $\lambda = 2\sigma_n^2$.

The cost function is a combination of a reconstruction term and a penalty term. The penalty term ensures that a sparse solution will be selected, and the reconstruction term ensures that the code will give a good representation of the signal. These two terms are opposing forces: a very sparse code cannot usually give a good representation, and vice versa. λ is a critical parameter that sets the balance between sparseness and reconstruction error.

The minimum of equation 2.24 can be found with a *gradient based* approach. The gradient of the error function with respect to the code is

$$\frac{\partial E}{\partial \bar{a}} = -2\mathbf{\Phi}^T(\bar{x} - \mathbf{\Phi}\bar{a}) + \lambda S'(\bar{a}) \tag{2.25}$$

MP and the gradient based approach are related; both approaches minimize the same cost function when $\lambda = 0$. However MP will find a sparse solution as it only adds one element at a time to the code, whereas the gradient based approach will not find a sparse solution as it can employ every code element in order to minimize the cost function. The penalty term $\lambda S(\bar{a})$ is therefore crucial to ensure a sparse solution for the gradient based approach.

2.4.2.1 SETTING THE RECONSTRUCTION ERROR-SPARSENESS BALANCE

λ is related to the noise present in the model. This noise is general, it incorporates any noise source: either noise in the signal and/or noise in the representation. There are various ways to choose the optimal value of λ and thereby set the noise level (Rao, Engan, Cotter, Palmer and Kreutz-Delgado, 2003). In a signal representation problem, λ is chosen so as to ensure a minimum signal-to-noise ratio (SNR). This approach is termed *quality-of-fit*; λ is simply set according to a prescribed SNR.

In a compression problem, the number of nonzero entries in the code is predetermined. For a given signal λ is then set such that the code has the preset number of nonzero entries. With this method λ has to be determined iteratively for each signal.

The *L-curve* method is another approach. Here λ is to set to some value that gives the best trade-off between the reconstruction error and sparseness. A plot of the reconstruction error versus the sparseness term for different values of λ is shaped like an "L". The L-curve theory states that the best choice for λ is the one that corresponds to the corner of the "L". The graph does not always have a definite corner, so it is proposed that the point of maximum curvature corresponds to the corner. This approach requires the model to be evaluated at various values of λ, making it impractical in most cases.

Ideally we would like to select that value for λ which will give the best performance of the entire system, in our case the best classification performance. This would require the entire system to be trained with several values of λ, which is not viable as the training of the entire system takes very long. Instead we use a different approach: λ should be chosen so that the reconstructed signal captures speech features that are important for speech recognition. Making λ too small will require the code to capture features that are irrelevant to speech recognition, while not all the important speech features will be captured when λ is too big. A proper value for λ can be estimated by visual inspection of several reconstructed signals at various values of λ.

2.5 FINDING THE OPTIMAL DICTIONARY

The dictionary has to capture the underlying statistical structure of the data for the code to be efficient. The optimal dictionary for a given dataset is difficult to find, but it is possible to find a good dictionary by using an iterative approach. This process is called dictionary *training*.

The following two sections show how to train a dictionary. The first section does so for codes that are selected by means of basis selection, the following section for codes that are selected in a probabilistic framework.

2.5.1 TRAINING A DICTIONARY FOR CODES FROM BASIS SELECTION

Here we derive a procedure that iteratively adapts the dictionary to make the codes that represent a set of stimuli more and more sparse.

The best code to represent a stimulus is the sparsest code that yields a reconstruction error less than a prescribed noise level. Generally, the sparser a code is the worse it reconstructs a signal. This means that the optimal code from equation 2.2 will have a reconstruction error that equals the prescribed noise level i.e. the constraint will be active.

Suppose the dictionary is adapted for a given signal and given code so that the reconstruction error is reduced. The adapted dictionary will allow a sparser code to be selected than the initial one, because the reconstruction error is smaller than the prescribed noise level; the code will have room to become sparser until the reconstruction error again equals the prescribed noise level.

This suggests a *sequential* procedure to adapt the dictionary. A change in the dictionary $\Delta\mathbf{\Phi}^{(n)}$ is determined for each signal $\bar{x}^{(n)}$ and associated code $\bar{a}^{(n)}$, one at a time. The dictionary is adapted with the term $\Delta\mathbf{\Phi}^{(n)}$ before the algorithm moves on to the next signal $\bar{x}^{(n+1)}$ in the stimulus set

$$\mathbf{\Phi}^{(n+1)} = \mathbf{\Phi}^{(n)} + \Delta\mathbf{\Phi}^{(n)} \tag{2.26}$$

The change in the dictionary $\Delta\mathbf{\Phi}^{(n)}$ should reduce the reconstruction error $R = \|\bar{x}^{(n)} - \mathbf{\Phi}^{(n)}\bar{a}^{(n)}\|^2$. The direction in which to change the dictionary is then

$$\Delta\mathbf{\Phi}^{(n)} = -\eta\frac{\partial R}{\partial\mathbf{\Phi}} \tag{2.27}$$

with η a learning parameter. The update formula is now

$$\Delta\mathbf{\Phi}^{(n)} = -\eta\left[\bar{x}^{(n)} - \mathbf{\Phi}^{(n)}\bar{a}^{(n)}\right]\bar{a}^{(n)T} \tag{2.28}$$

where $\bar{a}^{(n)T}$ is the transpose of $\bar{a}^{(n)}$.

The sequential update rule is difficult to perform in parallel, as a single $\bar{a}^{(n)}$ is determined for an update of the dictionary. It would be easier to parallelize a *batch* version of the update equation. In the batch version all the codes for the stimulus set is first determined for a given dictionary, only then is the dictionary updated.

For very small values of η, the batch version of the update formula approximates the sequential version. The batch update version can compactly be written as

$$\Delta\mathbf{\Phi} = -2\eta(\boldsymbol{X} - \mathbf{\Phi}\boldsymbol{A})\boldsymbol{A}^T \tag{2.29}$$

where the nth stimulus is the nth column in matrix $\boldsymbol{X}$, and the code that represents stimulus n is the nth column of matrix $\boldsymbol{A}$. This update formula is also used by Perrinet (2004b) to train dictionaries on codes from MP. Perrinet (2004b) notes that it is necessary to introduce a mechanism to ensure that the choice of any one dictionary element is not favoured above any other element. One way to achieve this requirement is to set the norms of all dictionary elements equal to a preset value.

2.5.2 TRAINING A DICTIONARY IN A PROBABILISTIC FRAMEWORK

The same arguments used in the previous section to find the optimal code can be used to find the optimal dictionary. In the previous section we were only interested in finding the optimal code, accordingly we use the notation $p(\bar{x})$. However, the probability distribution of signals arising from the model is also function of the dictionary. It follows that the Kullback-Leibler divergence is a function of the dictionary

$$KL(\mathbf{\Phi}) = \int_{\bar{x}} p_{true}(\bar{x}) \ln\frac{p_{true}(\bar{x})}{p(\bar{x} \mid \mathbf{\Phi})}d\bar{x} \tag{2.30}$$

The optimal dictionary is then

$$\mathbf{\Phi}^* = \max_{\mathbf{\Phi}} \langle \ln p(\bar{x} \mid \mathbf{\Phi}) \rangle \tag{2.31}$$

We approximate $\ln p(\bar{x} \mid \mathbf{\Phi})$ by sampling at its maximum posterior (see the discussion leading to equation 2.19)

$$\mathbf{\Phi}^* = \min_{\mathbf{\Phi}} \langle \min_{\bar{a}} E(\bar{a} \mid \bar{x}, \mathbf{\Phi}) \rangle \tag{2.32}$$

The optimal dictionary is found by integrating over all possible signals. If the distribution of the data set reflects the true distribution we try to model, then

$$\langle \min_{\bar{a}} E(\bar{a} \mid \bar{x}, \mathbf{\Phi}) \rangle \approx \sum_n \min_{\bar{a}} E(\bar{a}^{(n)} \mid \bar{x}^{(n)}, \mathbf{\Phi}) \tag{2.33}$$

Training of the dictionary is an iterative process that involves two steps: firstly the optimal codes for the stimulus set is found (solve $\min_{\bar{a}} E(\bar{a} \mid \bar{x}, \mathbf{\Phi})$), then the dictionary is adapted. Training can stop when there is not much change in the dictionary from one iteration to the next.

The dictionary can be adapted by a gradient based method. The cost function is $E_T = \sum_n E(\bar{a}^{*(n)} \mid \bar{x}^{(n)}, \mathbf{\Phi})$, for which the gradient with respect to the dictionary is

$$\frac{\partial E_T}{\partial \mathbf{\Phi}} = \sum_n \frac{\partial E(\bar{a}^{*(n)} \mid \bar{x}^{(n)}, \mathbf{\Phi})}{\partial \mathbf{\Phi}} \tag{2.34}$$

$$= \sum_n 2(\bar{x}^{(n)} - \mathbf{\Phi}\bar{a}^{*(n)})\bar{a}^{*(n)T} \tag{2.35}$$

This can be compactly written as

$$\frac{\partial E_T}{\partial \mathbf{\Phi}} = 2(\boldsymbol{X} - \mathbf{\Phi}\boldsymbol{A})\boldsymbol{A}^T \tag{2.36}$$

which is in the same form as equation 2.29.

We mentioned previously that this approximation leads to an optimal dictionary that has dictionary elements whose norm approach infinity. To see this consider that the optimal dictionary will be one that maximize $\langle \ln p(\bar{x} \mid \mathbf{\Phi}) \rangle$. The term $p(\bar{x} \mid \mathbf{\Phi})$ is approximated by the peak $\bar{a}^*$ of $p(\bar{x} \mid \bar{a}, \mathbf{\Phi})p(\bar{a})$. The maximum possible value that $p(\bar{x} \mid \bar{a}, \mathbf{\Phi})p(\bar{a})$ can reach will be a product of the maximum value of $p(\bar{x} \mid \bar{a}, \mathbf{\Phi})$ and the maximum value of $p(\bar{a})$. If the dictionary elements have norms that approach infinity, then the $\bar{a}$ that maximize $p(\bar{x} \mid \bar{a}, \mathbf{\Phi})$

will be close to the all zero vector, but the prior $p(\bar{a})$ also has its peak at the all zero vector. therefore the optimal dictionary is the one for which the dictionary elements have norms that approach infinity.

The trivial solution to the optimal dictionary does not exist if the norm of the dictionary elements is constrained. The constraint can be enforced strictly by fixing the norm of each dictionary element to a specific value (Kreutz-Delgado *et al.*, 2003), or it can be a soft constraint by adapting the norm of each dictionary element so the code element associated with that dictionary element has a prescribed variance.

Olshausen and Field (1997) use the soft constraint. They scale the norm of each dictionary element l_i after each iteration according to $l_{i,new} = l_{i,old}\left[\frac{\langle a_i^2\rangle}{\sigma_{goal}^2}\right]^{\alpha}$ with $0 < \alpha < 1$. When α is set too small, it will take a long time for the dictionary norms to converge. On the other hand if α is set too large, some dictionary elements may never learn useful structure. This will happen because the training algorithm only adapts elements to the extend that they are used by the code. If it so happens that a particular element is not used often early in the training process, it will initially have a small variance. This in turn will reduce its norm. An element with a small norm requires a relative large code value to make a significant contribution to reconstructing the signal. An element with a small norm is therefore less likely to be selected for a code since the sparseness function for large code values will penalize it. As it is not selected often its variance reduces even further. This leads to a point where the element is never selected as it did not learn any structure and it has a very small norm. We therefore choose to use the hard constraint as it avoids the problem of setting α.

The hard- and soft constraint approaches to bounding the dictionary norms will yield suboptimal dictionaries. A dictionary that is closer to the optimal dictionary can be found when the term $p(\bar{x} \mid \mathbf{\Phi})$ is not approximated by the peak $\bar{a}^*$ of $p(\bar{x} \mid \bar{a}, \mathbf{\Phi})p(\bar{a})$, but when the volume of the integral around the peak is estimated (Lewicki and Olshausen, 1999; Lewicki and Sejnowski, 2000). The volume estimation is unfortunately expensive to compute.

2.6 SPARSE CODING FOR SPEECH RECOGNITION

Speech recognition systems, which includes the brain, forms multichannel representations of raw speech waveforms. They can also handle signals of variable length. The sparse coding model discussed so far finds a sparse code only for a single channel signal of fixed length. In this section the simple LGM model is expanded to take multichannel signals of variable length as input.

2.6.1 SPEECH REPRESENTATION

We use the TIDIGITS (Leonard and Doddington, 1993) data set of continuous spoken digits by male and female speakers. The utterances are of variable length, consisting of a variable number of random digits. There are 11 different spoken digits, one for each number from "one" to "nine", a "zero" and an "oh". We choose this data set because it is a rather simple set: it has a limited vocabulary and simple language model (since the probability that a certain word follows any other word is approximately equal for all the words).

The label data supplied with TIDIGITS is limited to orthographic transcriptions. That is, the digit sequence of each utterance is given, but the start- and endpoints of each digit in the waveform are not specified. We estimated these points by doing forced alignment with the *HVite* tool that is part of the *Hidden Markov Model Toolkit* (HTK) (University of Cambridge, 2006). During forced alignment, HVite aligns a transcription with a waveform. The automatic forced alignment is accurate as HTK is able to model the data well: it is able to recognize almost all words in the test set correctly (WER[1]=3%).

It is time consuming to train sparse coding models. We therefore use a reduced data set so that the system can be trained in reasonable time. (With the reduced set, the entire system is trained in a week on sixteen Pentium 4 PCs working in parallel). The training set suggested for the TIDIGITS data set has 8623 utterances and contains 28329 words. Our training set is made up of every second sample of the suggested data set. The reduced set contains at least 2000 samples of every word, and is therefore sufficient for this initial investigation.

[1]Word Error Rate (WER) $= \frac{\#\text{deletions}+\#\text{insertions}+\#\text{replacements}}{\text{number of words}} 100\%$

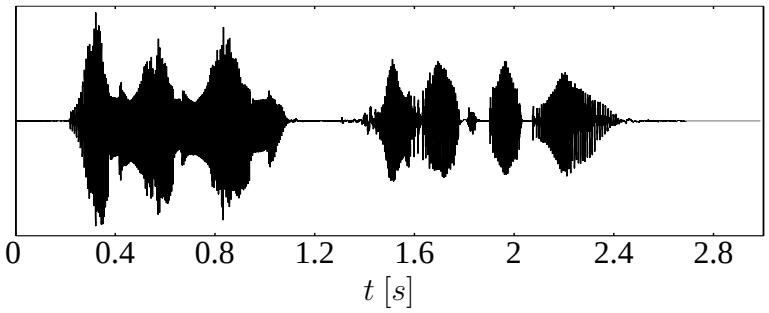
The simplest way to represent a raw speech signal is as a time waveform. However, time domain representations of speech are not as natural as frequency domain representations, as is evidenced by the fact that the auditory pathway uses a frequency domain representation. Even so, sparse coding of the time waveform of speech signals has shown some correspondence with physiological data (Lewicki, 2002). The auditory pathway forms a complex nonlinear representation of auditory stimuli. The representation produces phenomena like forward masking and two-tone suppression. There are models that provide frequency domain representations that to some extent agree with physiological measurements along the auditory pathway (Hermansky and Morgan, 1994; Wang and Shamma, 1995); however, for this study we choose to use a simpler spectrogram representation. The spectrogram we use is constructed by first dividing the raw signal into segments. The segments span 25.6ms (512 points with a sampling frequency of 20kHz). A new segment starts every 20ms or every 400 points, which gives an overlap between adjacent segments of 5.6ms. Each segment is then windowed with a Hanning window before the log magnitude of the Fast Fourier Transform for that segment is calculated. The magnitude is chosen because the human ear is insensitive to phase information; the log of the magnitude is used because our perception of sound intensity is often approximated by the logarithm although we actually perceive sound intensity on a cube-root scale (Hermansky, 1990).

Humans do not perceive the content of a signal on a linear frequency scale. We therefore use a Mel-spaced filter-bank to incorporate this nonlinearity. Each filter is a triangular bandpass filter whose centre frequencies are linearly spaced on a Mel-scale. Current automatic speech recognition systems use up to 32 filters, but we use only 8 filters for computational reasons. A small number of filters is preferred because too many filters put significant computational strain on the algorithms, and accurate temporal information is apparently more important than accurate spectral information for speech recognition (Allen, 1994; Kral, 2000; Moller, 1999; Shannon *et al.*, 1995). The outputs of the filters (with centre frequencies as given in figure 2.6) are then scaled so that the variance of each filter output over the data set is one.

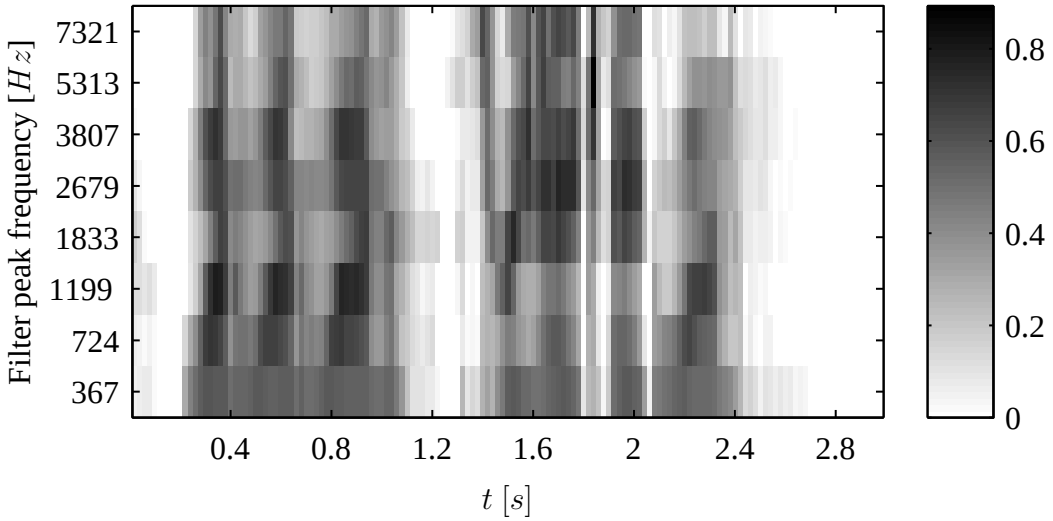
We use a linear generative model to find the sparse code of a spectrogram. This model works best if important speech features have large values in the representation and if si-

lences have values equal to zero. From inspection it seems for this particular data set and the spectrogram transform described above, that values below $15dB$ do not carry important speech features. We therefore subtract $15dB$ from the spectrogram after filtering and set any negative value equal to zero.

Figure 2.6 gives the signal representation for the utterance "one one one three eight eight one". The spectrogram representation $\boldsymbol{x}$ is an $N_c \times N_t$ matrix with N_c the number of channels or filters and N_t the number of segments. It serves as input to the linear generative model.



(a)



(b)

Figure 2.6: (a) The speech waveform for the utterance "one one one three eight eight one", and (b) its spectrogram representation. The gray scale indicates the signal amplitude $[dB]$.

2.6.2 TEMPORAL LINEAR GENERATIVE MODEL

The linear generative model applies to signals of a fixed length. Speech signals of variable length can be modelled with a LGM if the signal is segmented into fixed length segments. The code for each segment is then solved independently of other segments and the code for the complete signal is constructed by concatenating the codes from various segments to yield a temporal code. Such a code will have redundancies because the correlation between the code elements of neighbouring segments will be high, making the code less efficient.

A better way to model signals of variable length is to explicitly model the temporal code (Olshausen, 2002; Smith and Lewicki, 2005). The LGM now becomes a temporal linear generative model (TLGM). The reconstructed signal (in our case a spectrogram) $\boldsymbol{x}_R$ is a convolution of the code $\boldsymbol{a}$ with the dictionary $\boldsymbol{\Phi}$. An element at time t in channel c of the reconstructed spectrogram is found with:

$$x_{Rc,t} = \sum_{d=1}^{N_d} \sum_{T=1}^{N_t} \Phi_{c,\Delta t}^{\{d\}} \, a_{d,T} \tag{2.37}$$

For convenience we use the integer index t to correspond to a segment in the spectrogram. The index t is related to actual time by multiplying t with $20ms$. N_t is the number of segments in the spectrogram that is being reconstructed. d is an index to a dictionary element; there are $N_d = 32 \times 7 = 224$ dictionary elements in our dictionary (we will explain why this number is chosen below). $\boldsymbol{\Phi}^{\{d\}}$ is an $N_c \times \Delta\Phi$ matrix representing the dth dictionary element; $\Phi_{c,\Delta t}^{\{d\}}$ refers to the element in $\boldsymbol{\Phi}^{\{d\}}$ located in channel c at time Δt. N_c is the number of channels in the signal, for the spectrogram used here $N_c = 8$. $\Delta t = \Delta\Phi - T + t$ where $\Delta\Phi$ is the temporal width of a dictionary element. The width is taken as $260ms$ (or 13 segments), as evidence points to auditory memory being around $250ms$ in duration (Hermansky, 1998; Huggins, 1975; Massaro, 1972). When T is smaller than the temporal width of a dictionary element, the first $\Delta\Phi - T$ columns of the dictionary element are truncated. $\boldsymbol{a}$ is therefore an $N_d \times N_t$ matrix. We will refer to $a_{i,j}$ as a code element.

We now also extend the sparseness function to take a matrix $\boldsymbol{a}$ as an argument

$$S(\boldsymbol{a}) = \sum_{d} \sum_{T} S(a_{d,T}) \tag{2.38}$$

This sparseness function assumes that the activity of any two code elements is independent. It assumes that the activity of a channel is independent over time, and that the activity of different channels is independent. The assumptions fit the requirements of an efficient code.

2.6.3 FINDING THE SPARSE CODE OF A TLGM

For most practical problems the dimensionality of an LGM is small enough so that a sparse code can be found quickly. However the dimensionality of a TLGM is too large to find a sparse code in reasonable time. For example, a code of 40 dictionary elements that represents a $1s$ spectrogram of $20ms$ segments would yield a $40 \times 50 = 2000$ dimensional problem. A pure gradient based algorithm would take too long to converge to a sparse code. We need a more efficient way to find the sparse code.

The error function is a quadratic function of the code when the sparseness function is a quadratic function of the code. We can choose the sparseness function to be quadratic

$$S(a) = a - \frac{\beta}{2}a^2 \tag{2.39}$$

with $\beta > 0$. This is a valid sparseness function for $0 \leq a \leq 1/\beta$. The lower bound is consistent with a neural code that cannot have negative spikes; it is also required for this particular sparseness function to be valid (the sparseness function is negative for negative code elements). The upper bound is the point where a maximizes the sparseness function. The sparseness function decreases for values beyond this point, which does not fit a sparseness function. We found in our application that the upper constraint is never active for $\beta = 0.1$ and therefore does not play an important role in the code selection. We use a bound constrained quadratic programming algorithm called MINQ (Neumaier, 1998) to minimize the error function; it is much more efficient than a pure gradient based algorithm.

Consider again the 2000 dimensional problem. The Hessian matrix that is used in the quadratic programming algorithm would be a 2000×2000 matrix which is too big for efficient solution. Here follows an algorithm that does not require the entire Hessian.

We expect very few nonzero components in the solution, because we are looking for a sparse code. It is therefore not necessary to include all the components in a search for

the solution. In fact, it is only necessary to include those components that are likely to be nonzero. If it is possible to know these components beforehand, it will greatly reduce the dimensionality of the optimization problem.

However it is difficult to know which components will be nonzero, so we make use of an iterative process that chooses a small subset of components (size N_{set}) to be optimized in each iteration. The subset is determined by selecting the N_{set} most "promising" components from a set of candidate components. A candidate component is either nonzero in the current iteration, or it has a negative gradient with respect to the error function E. This criterion does not select *all* the components that could reduce the error function, because the error function is nonconvex, but it does select a few of them. The components that are included in the subset are the N_{set} candidate components with the largest gradient magnitude. The process of selecting components and optimizing them iterates until the reduction in the error function falls below a preset threshold. We used a threshold of 10^{-4} and select subset of size $N_{set} = 200$ during each iteration. We refer to this algorithm as subset selection and quadratic programming (SSQP). Its pseudo code appears in algorithm 1.

Fan, Chen and Lin (2005) describe a similar approach for support vector machines. Their algorithm selects a subset of two elements at a time, however the subset may be larger than two elements (Liao, Lin and Lin, 2002). Blumensath and Davies (2006) have also used a subset selection approach to find the sparse code of a high dimensional problem. An important difference between SSQP and their work is that SSQP performs several iterations of subset selection, whereas their approach selects the subset only once. We could not use such a selection criterion, as a reasonable subset would be too large to quickly calculate the Hessian matrix (size $N_{set} \times N_{set}$) required by quadratic programming.

SSQP is a gradient based algorithm which attempts to find the minimum of the quadratic error function $E(\boldsymbol{a})$. E is possibly a nonconvex function. This implies that there may be several local minima. We expect that most of the code elements in the optimal solution will be zero as we are looking for a sparse code. It is therefore sensible to start the optimization with an initial guess where all the code elements are equal to zero instead of a random guess, since such a starting point is expected to be close to the optimal solution. Other large scale

Algorithm 1 SSQP: Subset selection and quadratic programming

Input: $\boldsymbol{a}_{start}$ and $\boldsymbol{\Phi}$

Output: $\boldsymbol{a}_{final}$

$q \leftarrow 0$

$\boldsymbol{\alpha}_q \leftarrow \boldsymbol{a}_{start}$

repeat

$s_{active} \leftarrow \left\{[i,j] \mid \alpha_{q[i,j]} > 0\right\}$

$s_{neg} \leftarrow \left\{[i,j] \mid \frac{\partial E}{\partial \alpha_{q[i,j]}} < 0\right\}$

$s_{candidate} \leftarrow s_{active} \cup s_{neg}$

sort $s_{candidate}$ in descending order of $\left|\frac{\partial E}{\partial \alpha_{q[i,j]}}\right|$

$s_{use} \leftarrow \{s_{candidate\,[k]} \mid k = 1, 2, 3, \ldots N_{set}\}$

$\overline{\alpha}_{use} \leftarrow \{\alpha_{q[i,j]} \mid [i,j] \in s_{use}\}$

$\overline{\alpha}_{new} \leftarrow \min_{\overline{\alpha}_{use}} E(\boldsymbol{\alpha}_q, \boldsymbol{\Phi})$ {using quadratic programming}

$\boldsymbol{\alpha}_{q+1} \leftarrow \boldsymbol{\alpha}_q$ but with the s_{use} components replaced by $\overline{\alpha}_{new}$

$q \leftarrow q + 1$

until $E(\boldsymbol{\alpha}_q, \boldsymbol{\Phi})$-$E(\boldsymbol{\alpha}_{q-1}, \boldsymbol{\Phi}) < 10^{-4}$

$\boldsymbol{a}_{final} \leftarrow \boldsymbol{\alpha}_q$

optimization techniques may also be used to find sparse codes (a typical problem has around 30 000 variables).

2.6.4 THE DICTIONARY

We stated earlier that there are certain benefits to using an overcomplete dictionary. How many dictionary elements constitute a complete dictionary of the TLGM? This question is equivalent to finding the minimum number of dictionary elements required to reconstruct a signal perfectly. Each dictionary element spans across all the channels, and the code can select dictionary elements to be used at any time. Therefore a signal $\boldsymbol{x}$ of size $N_c \times N_t$ can be perfectly reconstructed with a minimum of N_c dictionary elements. However we constrain the code elements to be nonnegative, so the minimum dictionary size to reconstruct any signal perfectly is $2 \times N_c$.

The dictionary elements of a trained dictionary could model phonemes, which are the smallest meaningful acoustic units, or it can model syllables, or even complete words. The dictionary training process is unsupervised, which means that a particular dictionary element may learn any unit of speech. In fact it would be interesting to see what type of features the trained dictionary elements model. There are 20 phonemes in the speech that make up the data set, 11 syllables and 11 words. It is difficult to find the best dictionary size. The dictionary should at least be complete so that the code can successfully represent a signal, and the size of the dictionary may be related to the basic speech units that make up the data set. We use a two times overcomplete dictionary, so there are 32 dictionary elements. This dictionary is large enough to capture at least all the phonemes present in the data set.

In speech there are certain features which can be stretched over time without changing the meaning of the speech. To reflect this in the model, time scaled versions of the basic dictionary elements are constructed. We construct six scaled versions of each basic dictionary element and append them to the dictionary. The dictionary now contains $N_d = 32 \times 7 = 224$ elements.

A dictionary element $\mathbf{\Phi}^{\{d\}}$ is scaled over time with a scaling matrix $\boldsymbol{M}_s$

$$\mathbf{\Phi}_{\{s\}}^{\{d\}} = \mathbf{\Phi}^{\{d\}} \boldsymbol{M}_s \tag{2.40}$$

where $\mathbf{\Phi}_{\{s\}}^{\{d\}}$ is the scaled version of basic dictionary element d. $s \in 0, 1, 2, \ldots 6$ where $s = 0$ represents the unscaled dictionary element that spans $260ms$, $s = 1$ corresponds to the dictionary element that is scaled to span $280ms$, etc. $\boldsymbol{M}_s$ is determined in such a way that that the jth column of $\mathbf{\Phi}_{\{s\}}^{\{d\}}$ (which spans $20ms$) is

$$\left[\mathbf{\Phi}_{\{s\}}^{\{d\}}\right]^{\langle j \rangle} = \frac{1}{0.02} \int_{0.02\cdot(j-1)}^{0.02\cdot j} \mathbf{\Phi}_{\{0\}}^{\{d\}}(\tau \frac{0.26}{0.26 + 0.02s}) \mathrm{d}\tau \tag{2.41}$$

$\mathbf{\Phi}_{\{s\}}^{\{d\}}$ is a matrix where each column is used to reconstruct a $20ms$ segment of a spectrogram. We cast $\mathbf{\Phi}_{\{s\}}^{\{d\}}$ into the continuous time domain by letting $\mathbf{\Phi}_{\{s\}}^{\{d\}}(\tau) = \left[\mathbf{\Phi}_{\{s\}}^{\{d\}}\right]^{\langle i \rangle} \quad \forall \quad \tau \in [0.02(i-1), 0.02i]$. Figure 2.7 illustrates how a dictionary element is stretched.

The dictionary that includes the scaled elements is

$$\mathbf{\Phi} = \left[\mathbf{\Phi}_{\{0\}}, \mathbf{\Phi}_{\{1\}}, \mathbf{\Phi}_{\{2\}}, \ldots, \mathbf{\Phi}_{\{6\}}\right] \tag{2.42}$$

$\mathbf{\Phi}_{\{0\}}$ is the basic dictionary which has 32 elements; $\mathbf{\Phi}$ therefore has $32 \times 7 = 224$ elements.

The dictionary may be scaled in many other ways, for example linear interpolation of the basic dictionary should also work.

2.6.5 TRAINING THE DICTIONARY

The code will be more efficient if the dictionary is trained to reflect the regularities of the data. We use a gradient based training approach. First the sparse codes are determined for a given dictionary, then the dictionary is optimized for the given codes. The optimization is done by means of gradient descent with a line search. The derivative of the total error function E_T with respect to the dictionary is used as a search direction. A golden section method then finds the minimum of the error function along the search direction. The pseudo code for the training of dictionary elements appears in algorithm 2.

During the training process the norm of the dictionary elements tend to grow without bounds. We address this by forcing all dictionary elements to have a norm of one. Every

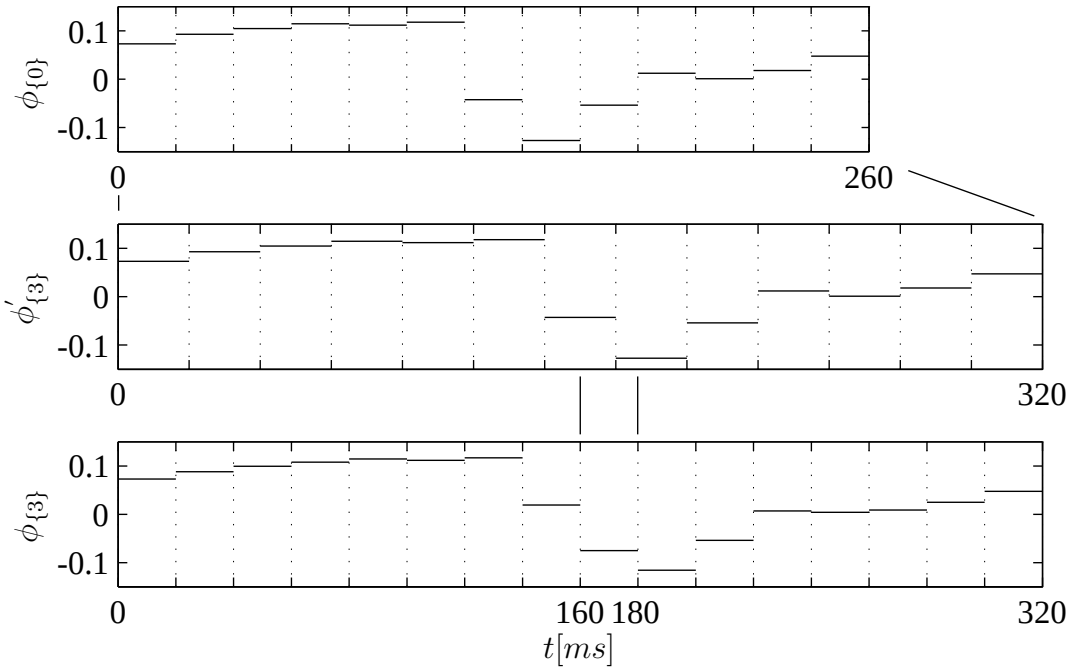


Figure 2.7: An unscaled dictionary element $\mathbf{\Phi}_{\{0\}}^{\{d\}}$ is a $N_c \times \Delta\Phi$ matrix. A row in the dictionary element corresponds to a channel in the spectrogram. This figure illustrates how a row of an unscaled dictionary element is scaled to $320ms$. $\phi_{\{s\}}$ refers to a row of the dth dictionary element $\mathbf{\Phi}_{\{s\}}^{\{d\}}$. The *top* plot shows a row of an unscaled dictionary element. The *middle* plot shows an intermediate step where the row is stretched to span $320ms$; $\phi'_{\{3\}}$ still has only 13 entries. The *bottom* plot shows that $\phi_{\{3\}}$ is created by resampling from $\phi'_{\{3\}}$. For example, the ninth entry which represents the interval $160ms$ to $180ms$, is the time averaged integral of $\phi'_{\{3\}}$ over that same interval. Every row of a dictionary element is stretched in the same way.

dictionary element of the initial dictionary is randomized. $\mathbf{\Phi}_{\{s\}}^{\{d\}}$ is a random $N_c \times \Delta\Phi$ matrix with entries sampled from a standard uniform distribution $U(0,1)$. We choose to use a distribution that will not give negative values. The motivation is that the dictionary elements are used to build up a representation that does not have any negative terms. Initially the elements does not have any structure so it will be unlikely to find a reasonable signal reconstruction by using elements that also have negative terms.

Algorithm 2 Training the dictionary

$k \leftarrow 0$

initialize $\mathbf{\Phi}_0$ as a random dictionary

normalize every dictionary element of $\mathbf{\Phi}_0$

repeat

for all n **do**

$\boldsymbol{a}_{k+1}^{(n)} \leftarrow SSQP(\boldsymbol{a}_k^{(n)}, \mathbf{\Phi}_k)$

end for

$\boldsymbol{A}_{k+1} \leftarrow \left\{ a_{k+1}^{(n)} \mid n = 1, 2, 3, \ldots N_s \right\}$

$\delta_{k+1} \leftarrow \min_\delta E_T(\boldsymbol{A}_{k+1}, \mathbf{\Phi}_k + \delta \frac{\partial E_T}{\partial \mathbf{\Phi}_k})$ {using a golden section search}

$\mathbf{\Phi}_{k+1} \leftarrow \mathbf{\Phi}_k + \delta_{k+1} \frac{\partial E_T}{\partial \mathbf{\Phi}_k}$

normalize every dictionary element of $\mathbf{\Phi}_{k+1}$

$k \leftarrow k + 1$

until $\mathbf{\Phi}$ has converged

It is computationally very expensive to train the dictionary. The reasons are that it is necessary to find the sparse code of every utterance at each iteration. The SSQP algorithm can take a long time to converge, especially if the starting point is far from the final solution. For example, when we want to find the sparse code to the utterance "one one one three eight eight one", we use a trained dictionary and set the starting point as the all-zero code $\boldsymbol{a}_{start} \leftarrow \mathbf{0}$ in the SSQP algorithm. This utterance is $2.6s$ long; it takes 121 iterations of SSQP to find the sparse code which has 172 nonzero elements. On an Intel Core2 1.86GHz PC it takes $88s$ for SSQP to converge.

To speed up the training process, we set the starting point of each training iteration as the solution to the previous training iteration $\boldsymbol{a}_{start} \leftarrow \boldsymbol{a}^k$. In this case it usually takes less than

five SSQP iterations to find the sparse code. It is only for the first iteration that we set the starting point as $\boldsymbol{a}_{start} \leftarrow \mathbf{0}$.

2.7 RESULTS

2.7.1 TRAINING THE DICTIONARY

It is computationally expensive to train the dictionary. The most expensive step is to find the sparse code of each utterance. We performed the coding step in parallel on sixteen Pentium 4 PCs. The code for each utterance can be found independently of the other utterances which makes it easy to parallelize the coding step.

To further speed up the training process, we start with a bigger value of λ than the one we would like to use; the bigger λ, the fewer nonzero elements are in the code and the quicker it is solved.

Initially the dictionary does not have any structure that corresponds to the data set. During the first few iterations the dictionary only has to be train on a few samples from the data set in order to learn some structure.

For the first 50 iterations we train the dictionary with $\lambda = 0.4$ and only on 1-in-5 samples of the data set. Thereafter we use $\lambda = 0.3$. For iterations 51 to 180 we train the dictionary on 1-in-5 samples. For iterations 181 to 245 we train it on 1-in-2 samples and from iteration 246 onwards on the entire data set (remember that our data set is not the entire TIDIGITS data set).

Figure 2.8 shows the performance of the training process on the *entire* data set, even though at times it is trained on a smaller data set. The error function is still decreasing at iteration 490 when the training is stopped. Further training should reduce the error function even more, but it appears if the improvement would not be significant. We also checked training progress against a validation set, and the cost function of the validation set is also still decreasing at iteration 490 (the validation results are not shown).

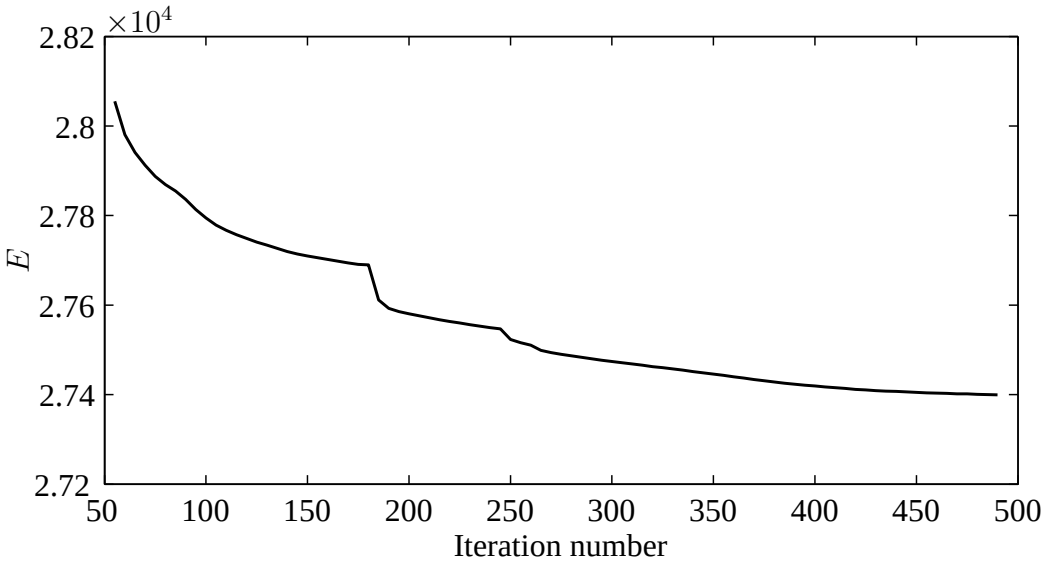


Figure 2.8: The history of the error function for the *entire* data set is shown in this figure. Refer to the text for more information on the training process.

2.7.2 PERFORMANCE OF CODE SELECTION ALGORITHMS

We tested the performance of three code selection algorithms using the trained dictionary: MP, SSQP and L-BFGS. L-BFGS is a limited memory BFGS algorithm (Byrd, Lu, Nocedal and Zhu, 1995). It is considered to be an efficient optimization algorithm for large-scale problems. The results are summarized in table 2.1.

MP is stopped when the value of the code element that is added to the code is less than 0.009. SSQP is stopped when the reduction in error function from one iteration to the next is less than 10^{-4} and L-BFGS stops when the norm of the gradient is less than 10^{-5}.

Of all the algorithms MP converges in the shortest time, but the solution is not nearly optimal. SSQP converges much faster than L-BFGS and the code it finds has the same error function value.

Notice that the value of the sparseness function for MP is more than that of SSQP, even though the MP code has fewer nonzero elements. This is a result of the sparseness function that we use; it is only a function of the code element value and not of the fact that an element

Table 2.1: The performance of three different sparse code selection algorithms. The input signal is “one one one three eight eight one”; the dictionary is trained with SSQP.

Algorithm[a]	$E(\boldsymbol{a})$	$R(\boldsymbol{a})$	$S(\boldsymbol{a})$	No. of nonzero elements	Time[b]
Matching pursuit (MP)[c]	25.399	13.907	38.305	80	17.7s
SSQP	12.268	1.890	34.594	162	34.4s
L-BFGS	12.268	1.860	34.700	164	320.9s

[a] All the codes are MATLAB 7 scripts.
[b] CPU time on an Intel Core2 1.86GHz PC.
[c] The MP algorithm does not make use of the sparseness function. The results for $E(\boldsymbol{a})$ and $S(\boldsymbol{a})$ are only given for comparison with other algorithms.

is zero or not.

The performance of an algorithm depends on the dictionary. The performance of MP will improve significantly when a MP-trained dictionary is used. However the results of L-BFGS should not change even if the dictionary is trained with L-BFGS. The reason is that L-BFGS finds very similar solution to SSQP as both algorithms are gradient based.

Table 2.2 gives the results to the same experiment as in table 2.1, but now the dictionary is trained on codes from MP. MP in the first row of the table is stopped when the code value that is added to the code is less than 0.008. This value is far too small, it allows the code to add elements that does not contribute significantly to the representation of the signal. It is better to stop MP when the SNR of the reconstructed signal falls below a predetermined threshold. MP$*$ is stopped as soon as the reconstruction error falls below that of the SSQP algorithm. We see that MP$*$ finds a code that reconstructs the signal just as well as SSQP, but it requires about half the number of nonzero code elements and finds the code in a fraction of the time it takes SSQP to converge.

The performance of L-BFGS for the MP-trained dictionary is again similar to the performance of SSQP. This emphasize the fact that these two algorithms are alike in their solutions, the only important difference is in the time it takes the algorithms to converge.

A comparison of tables 2.1 and 2.2 makes it clear that an algorithm performs much better with a dictionary trained by itself. We cannot use these results to determine which class of algorithm is better: basis selection or gradient based. One reason is that there are better basis selection algorithms than MP, such as *orthogonal matching pursuit* (Davis, Mallat and Avellaneda, 1997) and *optimized orthogonal matching pursuit* (Rebollo-Neira and Lowe, 2002). Another reason is that the application of the code is important. The code may be used as a means of signal compression or as a feature for pattern recognition.

Table 2.2: The same experiments as in table 2.1 but here the dictionary is trained on codes from MP.

Algorithm	$E(\boldsymbol{a})$	$R(\boldsymbol{a})$	$S(\boldsymbol{a})$	No. of nonzero elements	Time
MP[a]	24.086	1.767	74.395	400	44.4s
MP$*$	18.925	3.948	49.926	69	5.6s
SSQP	15.3082	3.988	37.735	108	42.2s
L-BFGS	15.0248	4.003	36.739	116	442.3s

[a]The MP algorithm does not make use of the sparseness function. The results for $E(\boldsymbol{a})$ and $S(\boldsymbol{a})$ are only given for comparison with other algorithms.

2.7.2.1 MP FOR PATTERN RECOGNITION

Matching pursuit is not a suitable algorithm to use when the sparse code it finds is a feature for pattern recognition. Robust pattern recognition requires that the feature representation does not change much if the input does not change much. Figure 2.9 shows the spectrograms of two utterances of “one six five seven”. There is not much difference between the spectrograms. The figure also shows the sparse code that MP finds for each utterance using the MP-dictionary; as well as the SSQP codes for each utterance using the SSQP-dictionary. The MP codes for the two similar utterances are dissimilar, while the two SSQP codes are similar. We found that this property of the MP codes also applies when MP codes are based on the SSQP-dictionary. When MP makes a poor selection of a code element in the first few

iterations, it cannot correct that mistake. This may be the primary reason that MP does not select robust codes.

2.7.3 PERFORMANCE OF SPARSENESS FUNCTIONS

Here we compare the performance of four sparseness functions for three different code selection algorithms. The sparseness functions[2] are l_0, $l_{0.5}$, l_1 and $S_{quad}(a) = a - 0.05a^2$. We base the performance on the signal-to-noise ratio (SNR) of the reconstructed signal and on the sparseness of the code.

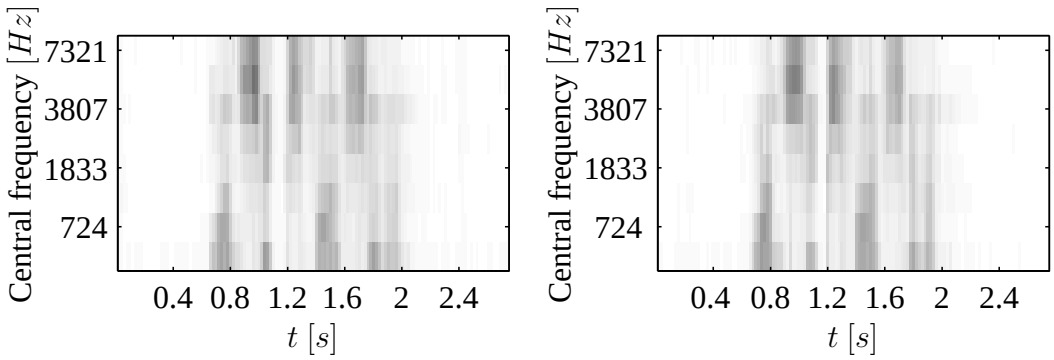
The code selection algorithms we use are MP, MP$^+$ and SSQP. MP$^+$ is the same as MP, except that it adds a final optimization step. A simple optimization can be performed on *any* given code that will not increase the l_0 norm while it may decrease the reconstruction error, or in the worst case leave the reconstruction error unchanged. It is the optimization of the values of the nonzero code elements. The values are changed so as to minimize the reconstruction error. MP$^+$ uses this final optimization step.

We train a dictionary for each of the algorithms. MP and MP$^+$ are implicitly trained on the l_0 norm as a sparseness function, SSQP is trained with S_{quad} as a sparseness function. The results appear in figure 2.10. Different points in the graphs are obtained by varying λ in the case of SSQP and the stopping threshold in the case of MP and MP$^+$. The points for which the dictionaries are trained are shown in each plot; SSQP is trained with $\lambda = 0.4$; MP and MP$^+$ with a threshold of 0.1. It is important to note that the SSQP dictionary we use to obtain figure 2.10 is not the dictionary we finally use to do speech recognition. Our final dictionary is trained with $\lambda = 0.3$. The exact value of λ is not important for the results of figure 2.10 but rather the trends in the figure.

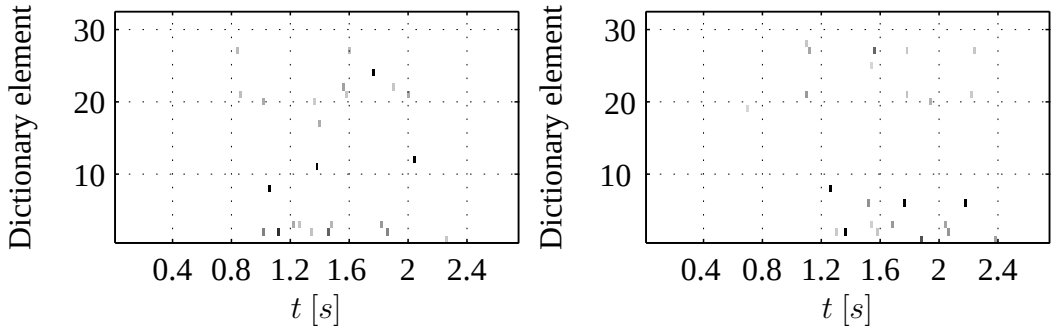
MP$^+$ performs best when the algorithms are compared using the l_0 norm, as can be expected. On the other hand, SSQP with S_{quad} performs best when the comparison is based on S_{quad}.

There is little difference between figures 2.10(c) and (d). The reason is that most of the

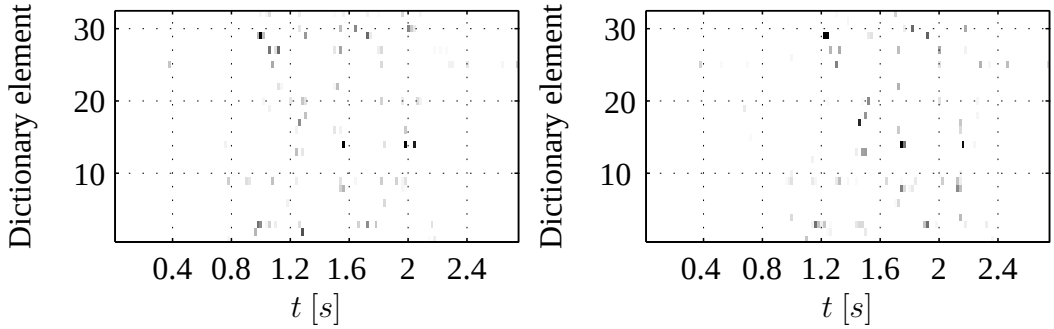
[2] $l_p(a) = |a|^p$. l_p is a valid sparseness function for $0 \leq p \leq 1$.



(a) Spectrograms.



(b) MP codes.



(c) SSQP codes.

Figure 2.9: (a) The spectrograms of two utterances of "one six five seven". Black represents $1dB$. (b) The compressed MP code for each utterance using the MP-trained dictionary. A code is compressed by adding the code elements associated with a particular dictionary element together irrespective of the scaling of that code element. The codes are compressed for better visualization. Black represents a code element value of 4.5. (c) The compressed SSQP code for each utterance using the SSQP-trained dictionary. Black represents a value of 1.2.

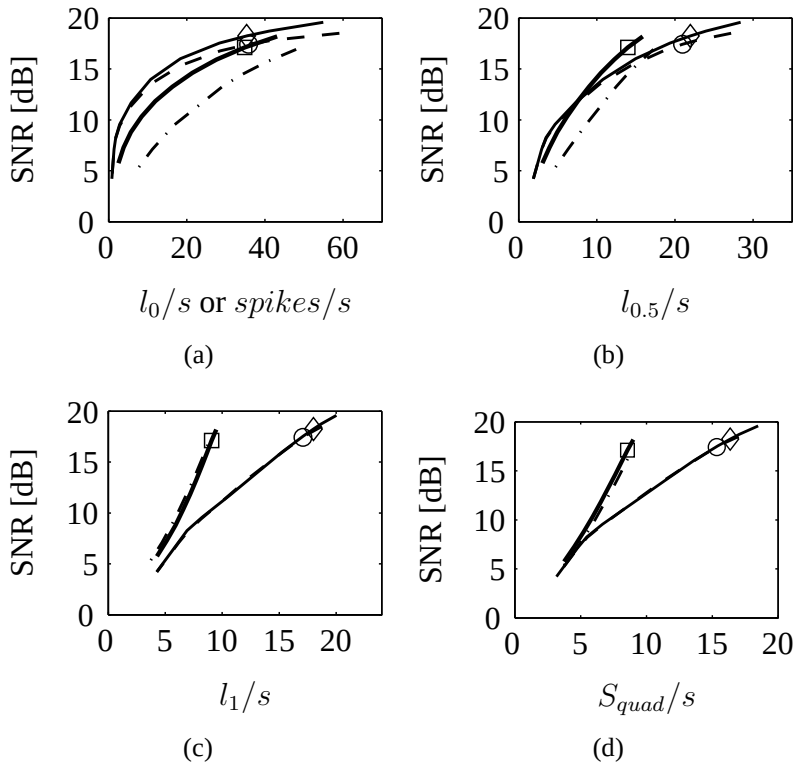


Figure 2.10: The four plots show the results for MP codes (thin dashed line), MP$^+$ codes (thin continuous line), SSQP codes obtained with S_{quad} as sparseness function (thick continuous line) and SSQP codes obtained with l_1 as sparseness function (thin chain line). The SSQP results are obtained with a dictionary trained on SSQP codes with the S_{quad} sparseness function, while MP and MP$^+$ use dictionaries optimized for each respective algorithm. The lines for MP and MP$^+$ in plots (c) and (d) are almost coincident. The x-axis is an average sparseness value per second, the y-axis is the SNR of the reconstructed signal for the dataset. A marker indicate that point on the plot where the stopping threshold or λ is the same as was used during training of the dictionary (circle for MP, diamond for MP+, square for SSQP).

code element values for MP and SSQP lie in that range where S_{quad} and l_1 are very close together. S_{quad} and l_1 have almost identical functional plots in this range ($a < 2$). It would seem at first that there is not any benefit to using the computationally more expensive S_{quad} when l_1 yields almost the same results. However from figure 2.10(a) we see that codes calculated with S_{quad} has far fewer nonzero components than codes calculated with l_0 for the same SNR. This property of the S_{quad} sparseness function is very desirable. Remember that ideally we want to use l_0 as a sparseness function, but cannot as MP yields codes that are not suited to pattern recognition. S_{quad} is a better sparseness function than l_1 in this respect.

2.7.4 THE TRAINED DICTIONARY

The basic dictionaries $\mathbf{\Phi}_{\{0\}}$ that are trained with MP and SSQP are shown in figures 2.12 and 2.11. Each dictionary has dictionary elements whose structure reveal complex features that span across frequency and time to varying degrees. It is clear that some elements are strongly localized in frequency and others more localized in time; several seem to encode specific transitions of frequency content as a function of time.

We see that the SSQP dictionary has more dictionary elements that code complex features. For example elements 12 to 16 and others of the MP dictionary are similar and are not as complex as element 2 for example. This is in contrast to the image dictionaries published by Perrinet (2004a) who used MP and Olshausen and Field (1996b) who use a gradient based approach. Comparing the MP image dictionary with the gradient based image dictionary shows that the MP dictionary has elements that are less localized in space and higher portion of elements that are high-frequency Gabors. We found that the gradient based approach has more localized elements than MP. The difference may be due to the different type of dataset we use.

We can use histograms of phoneme occurrences to form an idea of the sounds coded by the elements. A histogram is created for each phoneme p and each dictionary element d. The bins of the histogram span over time where each bin is $20ms$ wide. We associate a time period t_b with each bin; it is a period that precedes a spike associated with dictionary element d. The number of entries in bin b of histogram (p, d) is the number of times phoneme

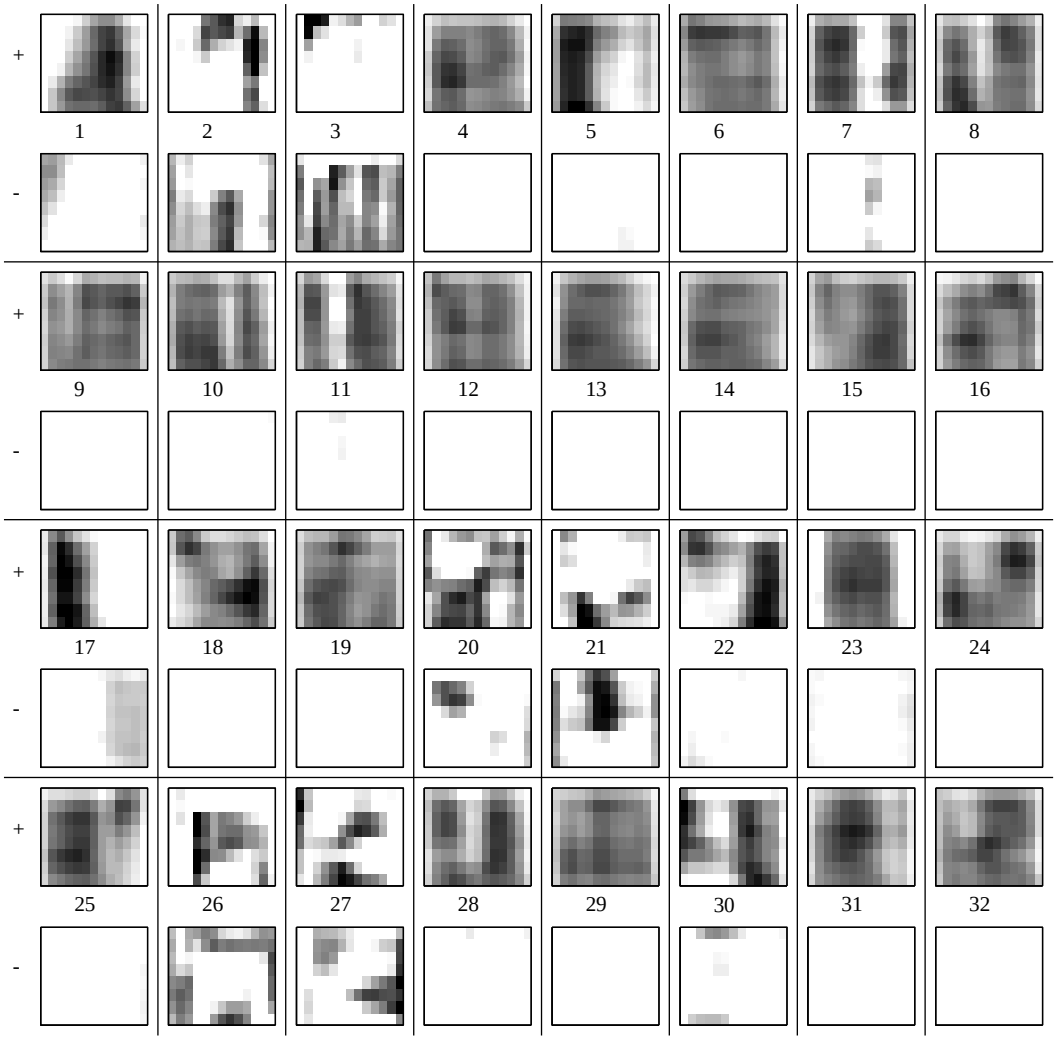


Figure 2.11: This figure shows the dictionary elements of the basic dictionary $\mathbf{\Phi}_{\{0\}}$ trained with MP. Each element is split into positive and negative parts for better visualization. A column of a dictionary element spans $20ms$; a complete dictionary element therefore spans $260ms$. Each row in a dictionary element correlates with a channel in the spectrogram (see figure 2.6). Thus, the bottom row of a dictionary element represents signal content at $367Hz$ and the top row $7321Hz$. The dictionary elements have a norm of 1. White represents values of 0 and black 0.2. Elements are numbered from left-to-right and from top-to-bottom.

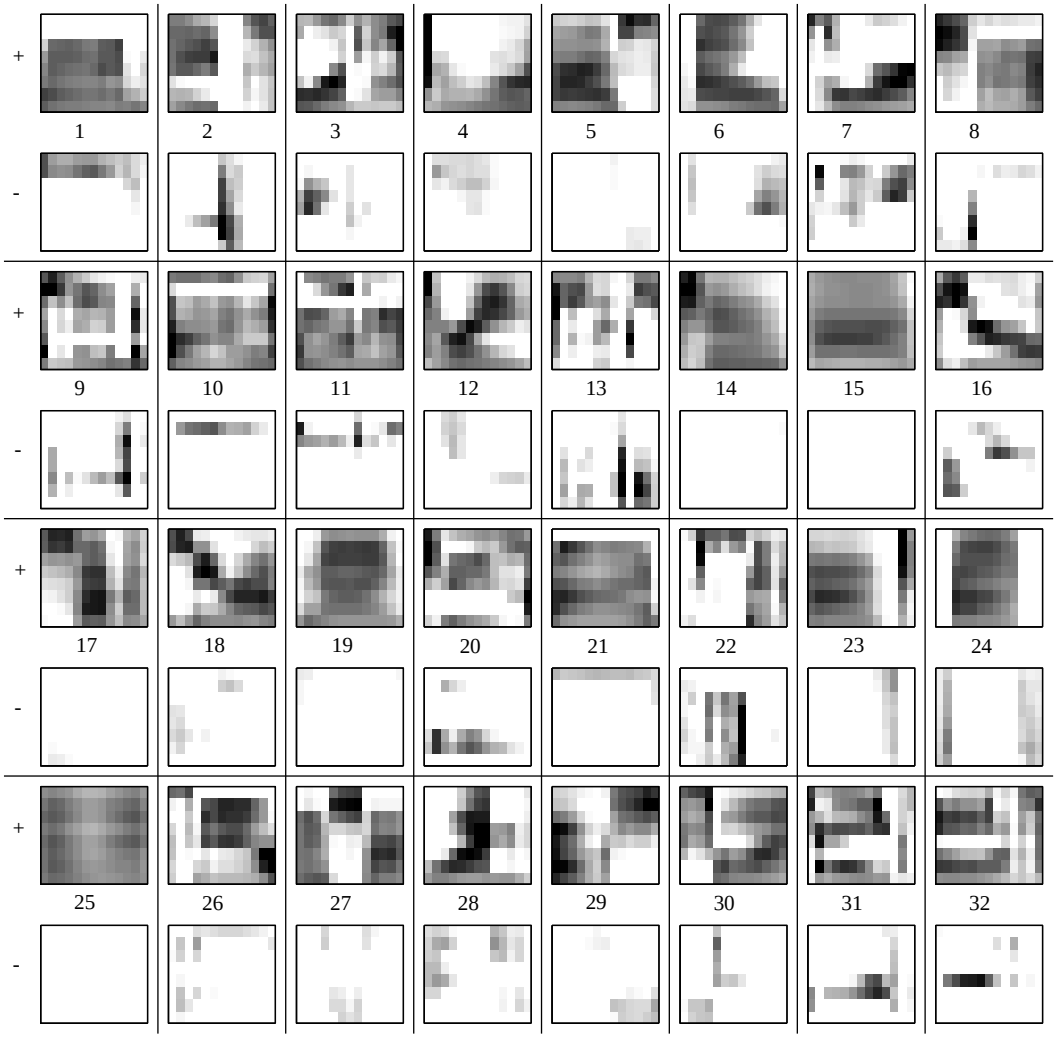


Figure 2.12: This figure shows the 32 dictionary elements of the basic dictionary $\mathbf{\Phi}_{\{0\}}$ trained with SSQP. Each element is split into positive and negative parts for better visualization. A column of a dictionary element spans $20ms$; a complete dictionary element therefore spans $260ms$. Each row in a dictionary element correlates with a channel in the spectrogram (see figure 2.6). Thus, the bottom row of a dictionary element represents signal content at $367Hz$ and the top row $7321Hz$. The dictionary elements have a norm of 1. White represents values of 0 and black 0.2. Elements are numbered from left-to-right and from top-to-bottom.

p precedes a spike of dictionary element d during t_b for the entire data set. In other words, the histogram shows how often and at what times a phoneme occurs before a spike. In the TLGM a spike in the code means that a dictionary element is used to reconstruct a part of the spectrogram that *precedes* the spike. (Note that we ignore the fact that time-scaled versions of a dictionary element may in fact have been used; all the scaled versions are grouped together and are viewed as a single dictionary element).

Figure 2.13 shows the histograms of phoneme occurrences for four dictionary elements. From figure 2.13(a) is appears that dictionary element $d = 2$ is used in coding the "ih-k-s" sound in "six". It also seems that it is used to code "ey-t" which occurs in "eight". It is not possible to conclude from this histogram alone that element $d = 2$ actually codes the "ih" sound. The reason is that in our limited dictionary the "ih" sound always precedes the "k-s" sound. If $d = 2$ codes only the "k-s" sound the histogram would still show that the "ih" sound often precedes a spike associated with $d = 2$. However, by taking a closer look at the dictionary element itself, we see that it has a period of silence between $80ms$ and $140ms$ before the spike occurs. This corresponds to the closure between the "ih" and "k" sounds. therefore it does seem that element $d = 2$ codes the "ih" sound or at least a part of it. There are certain implications to $d = 2$ having a large negative part. Our spectrogram transform has only positive parts, which means that $d = 2$ cannot be used on its own. It has to be used with other dictionary elements.

Figure 2.13(b) shows a dictionary element that codes a sound with a rising frequency. We see from the histogram that the entire word "three" often precedes a spike from $d = 12$. The plot of the dictionary element itself shows that it contains information to code the word "three". The "th" fits the left part of the dictionary element; it shows a broad distribution of energy across most of the frequencies. The right part of the dictionary element fits the rising centre-of-mass (in frequency) of the sound in "iy".

Dictionary element $d = 25$ (figure 2.13(c)) does not code a particular sound, but is rather used as a constant offset. Lastly figure 2.13(d) shows that $d = 31$ is a dictionary element with a very complex structure. The histogram shows that it is mostly used during the coding of "ey" in "eight".

There is good reason to believe that the basic unit of speech is on the syllable level (Nguyen and Hawkins, 2003). The dictionary elements are in good agreement with this, at least some of them (for example $d = 2$) are on syllable level. Dictionary elements are not used in isolation, they are used in conjunction with other elements. It is complicated to say which acoustic unit a particular dictionary element codes as it depends on the other dictionary elements that are used in with it. We see for example from figure 2.13(a) that $d = 2$ is used in coding both the words "six" and "eight" although the words does not share a phoneme.

2.7.5 THE CODE ELEMENTS

Figure 2.14 gives the histogram of all the spikes for the entire dataset. It does not show the code elements that did not spike. Only 0.0042% of the code elements are active. The distribution agrees with that of a sparse code where most of the code elements are zero. A great percentage of the code elements have values close to zero. This is unwanted feature is caused by the prescribed prior probability that has a peak for values close to zero. The figure shows the shape of the prescribed prior probability $p(a) \propto e^{-(a-0.05a^2)} \; \forall \; 0 \leq a \leq 10$. The shape of the histogram does not follow that of the prior. Two important reasons for this are that the dictionary elements are constrained to have a norm of one and that the underlying distribution of the data may not fit the prescribed prior. The shape of the histrogram shows that the computed codes are more sparse than the prior prescribes. This is not a problem as we are looking for sparse codes.

Actual data agrees much better with the prior when the optimization problem is formulated so that a constraint on the dictionary elements is not necessary (Girolami, 2001). Such a formulation unfortunately makes it computationally even more expensive to find a sparse code. Another reason for the difference between the data and the prior is that the prior only applies to a dictionary where every dictionary element is trained. In our dictionary only the basic unscaled dictionary elements are trained, not the scaled versions. The sparseness function is not chosen for its shape but the fact that it makes the error function quadratic, therefore the exact shape of the spike histogram is not that important to our application.

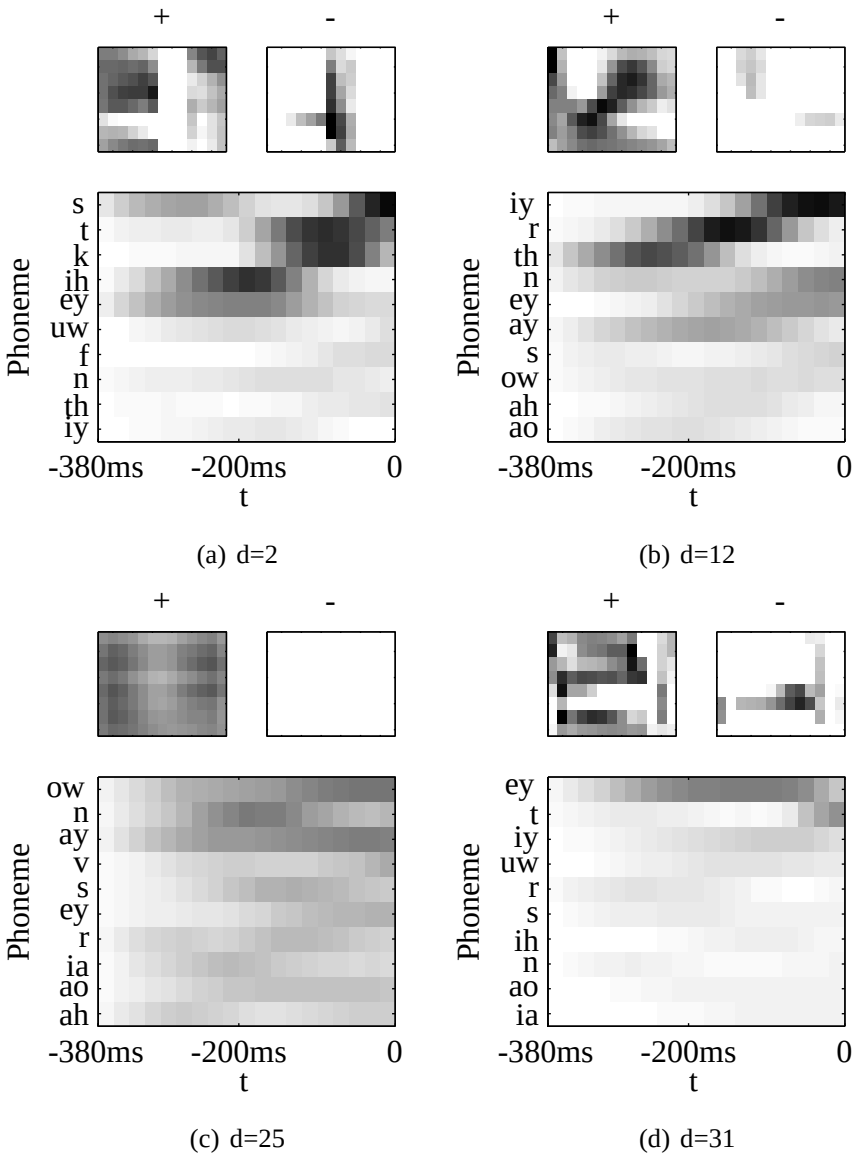


Figure 2.13: The top row of each plot gives the positive and negative parts of the dictionary element. The bottom row of each plot depicts the phonemes' correlation with the given elements: it is a histogram plot that shows which phonemes preceded the spike associated with that particular model, and at what times the phoneme occurred. We considered only spikes with an amplitude of more than 0.124; smaller spikes are not that significant. The histogram plot shows only the ten phonemes that occurred most for each dictionary element. The histograms are scaled so that black represents 2000 entries in a bin.

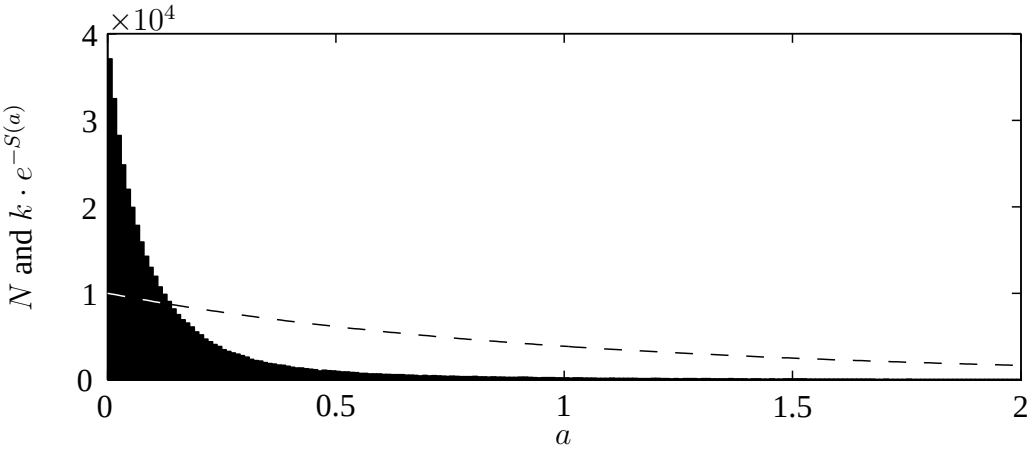
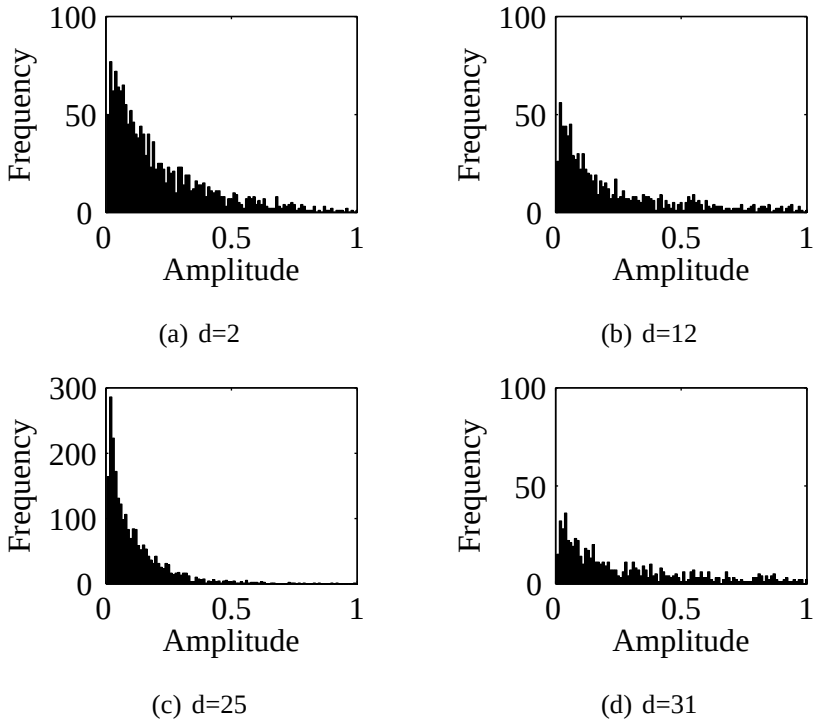


Figure 2.14: The histogram of spike values for the training set. It is gathered from the codes for the complete dictionary (includes all the scaled dictionary elements). The dashed line gives the shape of the probability density function $p(a) \propto e^{-S(a)}$. The histogram should ideally follow the shape of $p(a)$.

It is interesting to look at the amplitudes that spikes assume for a particular dictionary element. Figure 2.15 shows histograms of the spike amplitudes for various dictionary elements. We see that the dictionary elements are not used equally often; $d = 31$ is used less often than any of the other dictionary elements that are shown. This may be due to the complex and irregular structure of $d = 31$, we do not expect such a structure in speech. $d = 25$ is used often but has a small average spike value. The spike value indicates the contribution that a dictionary element makes to reconstruct a signal. The fact that $d = 25$ has a smaller average spike value than the other dictionary elements suggests that it too does not capture significant structure in the speech.

Figure 2.16 shows the frequency with which a given dictionary element is used, as well as the frequency with which a given scale is used. Dictionary element $d = 15$ is seldom used. This element seems to serve a similar purpose as $d = 25$ which is to provide an offset. However $d = 15$ has significant low frequency content which may be the reason its usage is different from $d = 25$. The figure also shows that the maximum scaling ($s = 6$) of dictionary elements are used more often than any other scaling. This is because the sparseness function promotes sparse codes; it is "cheaper" to increase the amplitude of a



(a) d=2 (b) d=12

(c) d=25 (d) d=31

Figure 2.15: The histograms of spike amplitudes for four of the dictionary elements are given. These are compiled from the sparse codes of the training set. Only nonzero spikes are included in the histograms. The histograms show that most of the spikes have very small amplitudes. We explain in the text that such spikes are not desirable. It also shows that the dictionary elements are not used equally often; $d = 25$ is used more often than any of the other three dictionary elements.

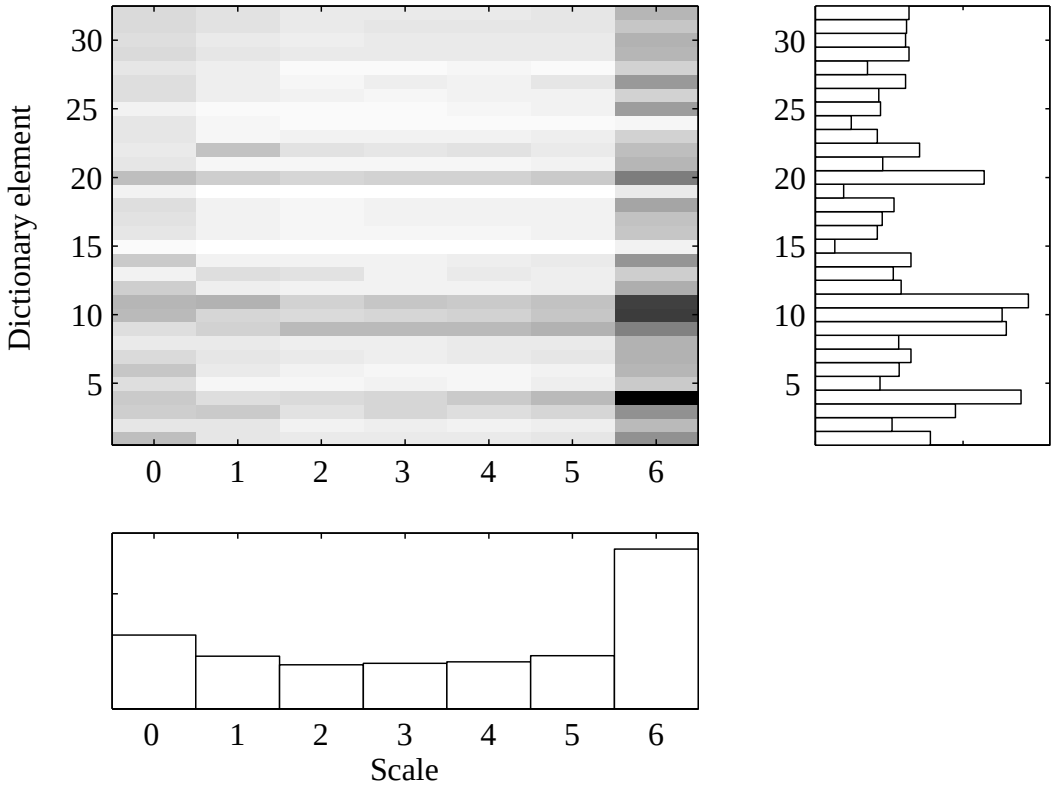


Figure 2.16: The *top left* plot consist of 32 histograms, each histogram is a row in the plot. A histogram shows how often a particular scale is used for the given unscaled dictionary element. The *bottom left* histogram shows the frequency with which a particular scale is used, independent of the unscaled dictionary element. The *top right* histogram gives the frequency with which a particular unscaled dictionary element is used, independent of scale.

spike in order to reconstruct a code than to use an additional spike. A code can consist of fewer nonzero entries if the dictionary elements span over long time periods than if the dictionary elements are temporally short.

Not all dictionary elements are most often used at the maximum scale. For example $d = 12$ is also often used unscaled. The reason for this is explained by the fact that $d = 12$ codes the entire word "three" (see figure 2.13(b). 43% of the occurences of "three" in the dataset are shorter than $280ms$ (the time span of an unscaled dictionary element). It makes sense that the unscaled version of $d = 12$ is used so often.

A code is efficient when the code elements are independent. Figure 2.17 shows the

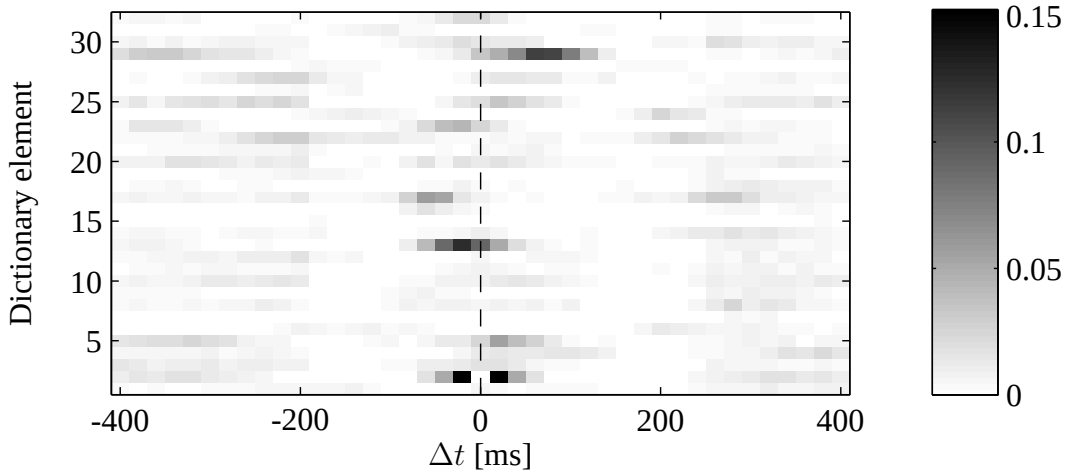


Figure 2.17: The probability that a spike at time t_i precedes or follows a spike of dictionary element $d = 2$ that occurs at t_2. The x-axis is the time relative to the $d = 2$ spike, $\Delta t = t_i - t_2$. The entry at $(d = 2, \Delta t = 0)$ is 1, but is removed to give a better range to the shades in the plot. The plot is constructed for spikes with an amplitude larger than 0.124 and *only* for spikes from the $s = 6$ scale (dictionary elements that span $380ms$).

correlation between spikes from $d = 2$ with other spikes. There is overall little correlation between spikes. A weak correlation exist between $d = 2$ and itself around $\Delta t = \pm 20ms$. This shows that one spike sometimes immediately follows another. One would not expect this to occur as often as it does. We closely inspected some of the codes that have two consecutive $d = 2$, $s = 6$ spikes, but could not find a solution for those particular codes that does not have the two consecutive spikes. Different starting points and different gradient based algorithms all yielded codes that have the two spikes. It therefore seems that the occurrence of two consecutive spikes is not an artefact of the optimization but rather a feature of the code.

Figure 2.17 also shows a weak correlation between $d = 2$, $d = 13$ and $d = 29$. This correlation exist because the combination of these dictionary elements are part of the reconstruction of the word “six”. This pattern can in fact be used to recognize the word “six” from a code.

2.7.6 THE IMPORTANCE OF SCALING DICTIONARY ELEMENTS

We use scaled versions of dictionary elements because in speech certain features may be stretched over time without changing its meaning. Therefore we do not expect useful information for speech recognition to be contained in the particular scaled version of a dictionary element that is used, i.e. we expect a spike associated with $\mathbf{\Phi}_{\{s\}}^{\{d\}}$ to signal the same event as a spike associated with $\mathbf{\Phi}_{\{s'\neq s\}}^{\{d\}}$.

Figures 2.18 and 2.19 shows whether there are any information useful for speech recognition contained in the scaling itself. These figures are constructed similarly to the histograms of phoneme occurrences in figure 2.13; they show how often a word precedes a spike. The difference is that here we correlate activity with words and not phonemes, we also create a different plot for each scale.

Figure 2.18 confirms that $d = 2$ is used most often to code "six" and "eight". We see for a particular scale that there is very little difference between the histograms of "six" and "eight". This means that scaling cannot be used to distinguish whether the spike signals a "six" or an "eight". The plot for $s = 6$ does show that scaling can be used to distinguish "two" from either "six" or "eight". We see that "two" is mostly used at the $s = 6$ scale. A classification system that uses the scaling information will classify "two" more accurately as it will use the information that a spike associated with $d = 2$ and $s = 0, 1, 2, \ldots 5$ probably does not signal a "two". It appears that in most cases scaling does not contain useful information, although scaling may be useful in other cases. Figure 2.19 leads us to a similar conclusion. It shows that a spike associated with $d = 12$ most probably signals a "three", independent of the scaling. However for $s = 6$ the spike may also signal an "eight".

Figures 2.18 and 2.19 suggest that scaling can be a good feature to use in speech recognition, although scaling adds very little information about the spikes.

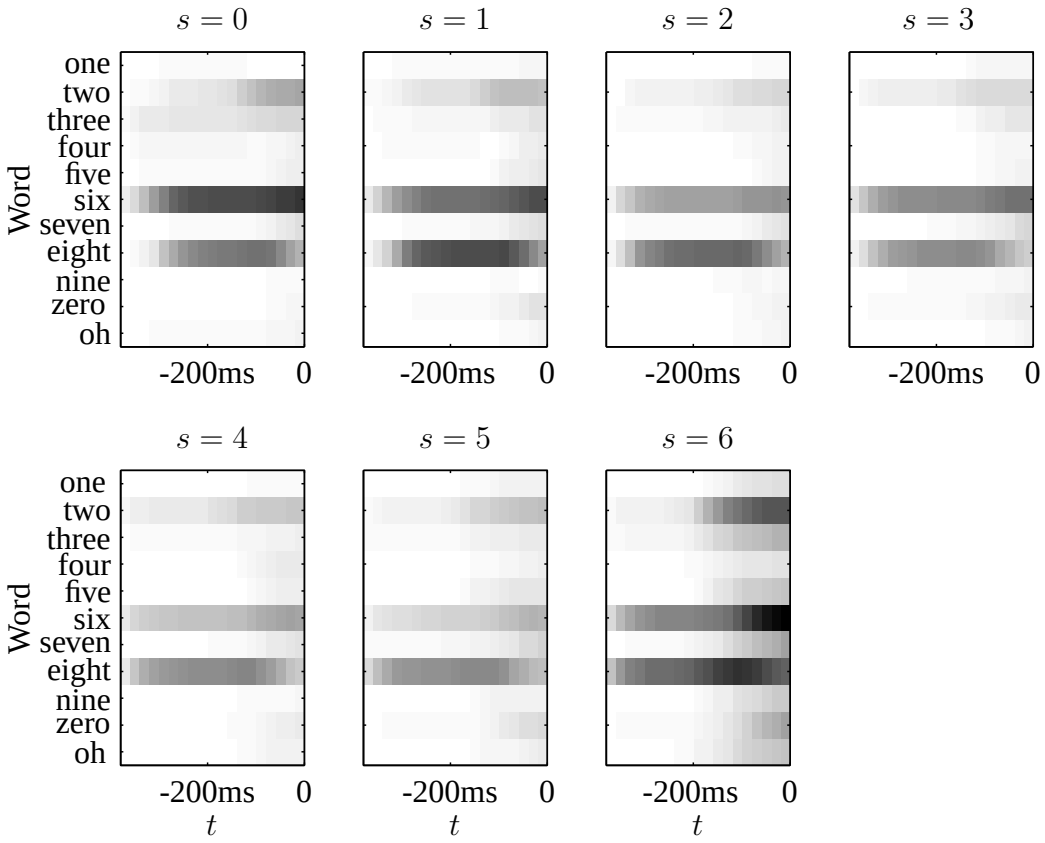


Figure 2.18: The correlation between words and spikes of $d = 2$. The intensity of an entry in the plot shows how often a particular word precedes a spike at a particular time. Each plot is constructed for a different scale. We only considered significant spikes (spikes with an amplitude larger than 0.124).

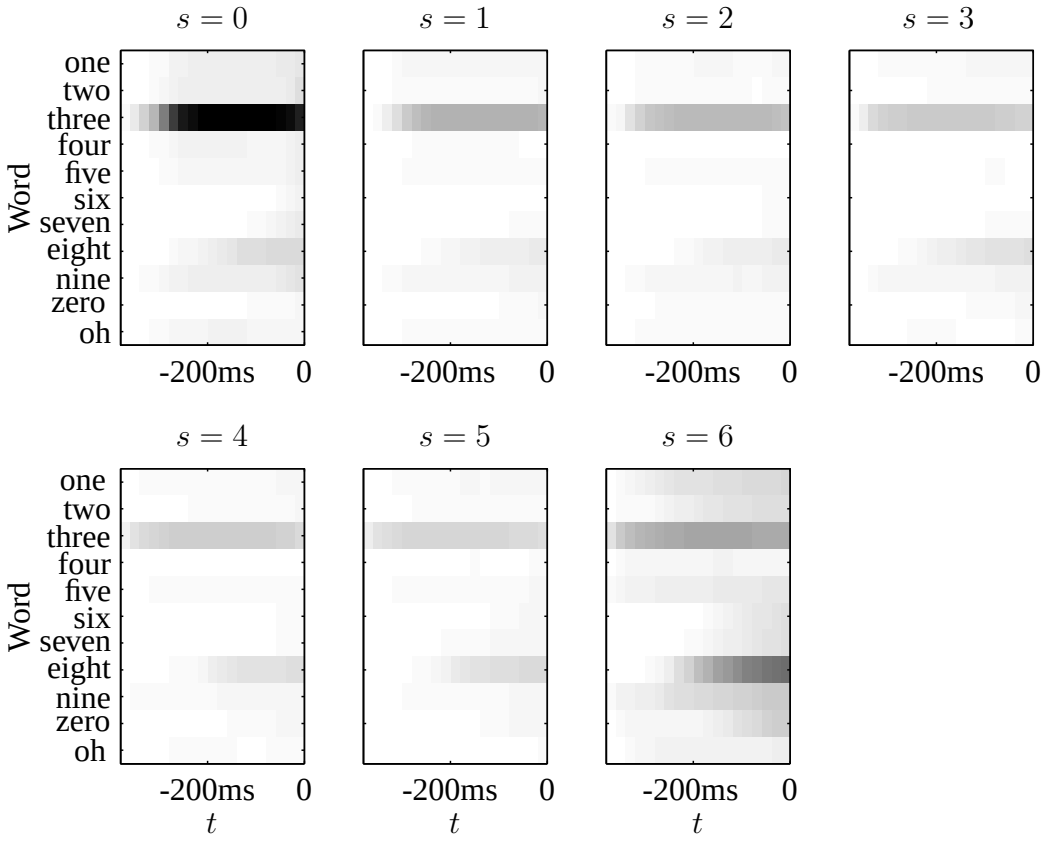


Figure 2.19: The same as figure 2.18, but for $d = 12$.