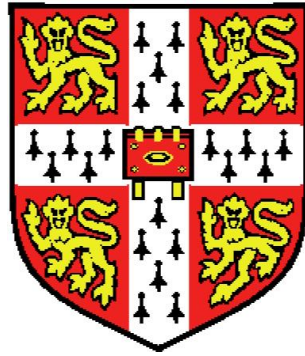


Distributed Associative Memory



Philip J. Sterne
Pembroke College
University of Cambridge

This dissertation submitted for the degree of
Doctor of Philosophy
November 2010

ABSTRACT

This dissertation modifies error-correcting codes and Bloom filters to create high-capacity associative memories. These associative memories use principled statistical inference and are distributed as no single component contains enough information to complete the task by itself, yet the components can collectively solve the task by passing information to each other. These associative memories are also robust to hardware failure as their distributed nature ensures there is no single point of failure.

This dissertation starts by simplifying a Bloom filter so that it tolerates hardware failure (albeit with reduced performance). An efficient associative memory is created by performing inference over the set of items stored in the Bloom filter. This architecture suggests a modification which forgets old patterns stored in the associative memory (known as a palimpsest memory). It is shown that overwriting old patterns in an independent manner reduces performance, but is still comparable to the well-known Hopfield network. The lost performance can be regained using integer storage which allows the superposition of the pattern representation, or ensuring bits are not overwritten independently using concepts from error-correcting codes. The final task performs recall in continuous time using components which are more similar to neurons than used in the rest of the dissertation. The resulting memory has the exciting ability to recall many patterns simultaneously.

Statistical inference ensures gradual degradation of the performance as an associative memory is overloaded. Since many definitions of associative memory capacity rely on the existence of catastrophic failure a new definition of capacity is provided.

In spite of some biologically unrealistic attributes, this work is relevant to the understanding of the brain as it provides high performance solutions to the associative memory task which is known to be relevant to the brain.

DECLARATION

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of work done in collaboration. No part of this dissertation has already been or is being concurrently submitted for any such degree, diploma or any other qualification at any other university. This dissertation does not exceed the word limit of sixty thousand words, including tables, footnotes, bibliography and appendices, set out by the Faculty of Physics and Chemistry.

Signed: _____

Philip Sterne
Cambridge
8th November 2010

ACKNOWLEDGEMENTS

This dissertation would not exist were it not for the clear insight and constant encouragement of David MacKay. I would also like to thank the members of the Inference group with whom I regularly interacted: Philipp Hennig, Carl Scheffler, Christian Steinruecken, Emli-Mari Nel, Keith Ver-tanen, Tamara Broderick, Ryan Adams, Seb Wills.

Outside the inference group I had numerous interesting discussions with Máté Lengyel (Engineering), Petra Vértés (Physics) and Jean-Pascal Pfister (Engineering), and this dissertation is definitely the better for it.

My funding came from Pembroke College, the Cambridge Commonwealth Trusts and my family, and I am grateful to have been given such an amazing opportunity.

Finally I must thank the college friends who kept me sane and ensured I had a life outside of statistical inference: Ed Brambley, Patrick De-whurst, Joey Hanzich, Jennifer Harcourt, Eric Jägle, Josh Karton, Katie McAllister, Johann Jakob Napp, Kelly Randell, Theresia Schaedler, Simon Schlachter, Dan Shouler, Desirae Vanek, and Csilla Várnai.

Contents

1	Introduction	1
1.1	Associative memory	3
1.2	Recall as inference	4
1.3	Capacity	5
1.4	Common themes in our solutions	5
1.5	Overview of the dissertation	6
2	The neurally-inspired Bloom filter	9
2.1	The Bloom filter	10
2.2	Neurally-inspired Modifications	13
2.2.1	Storage flexibility	18
2.3	Related Work	19
2.3.1	The Willshaw network	19
2.3.2	The Sigma-Pi associative network	21
2.3.3	Familiarity detection with Hopfield networks	22
2.3.3.1	Palimpsest familiarity detection	23
2.3.4	Related Bloom filters	24
2.3.5	The WISARD architecture	24
2.4	Results	25
2.4.1	Sensitivity to a single bit flip	27
2.4.2	Conclusion	28
3	Binary associative memory	31
3.1	Neurally-inspired associative memory	32
3.1.1	The sum product algorithm	32
3.1.2	Loopy Belief Propagation	36
3.1.3	Recall	36
3.1.4	Calculation of Messages	38
3.1.5	Theoretical estimate of the bit-error	40
3.1.6	Measuring Recall Performance	43

CONTENTS

3.1.7	Capacity of the neurally-inspired associative memory	45
3.2	Neurally-inspired palimpsest network	49
3.2.1	Storage in the palimpsest network	49
3.2.2	Retrieval	51
3.2.3	Capacity of the neurally-inspired palimpsest memory	53
3.2.4	Related Work	56
3.2.4.1	The Palimpsest Willshaw network	56
3.2.4.2	Stochastic storage rules	56
3.2.4.3	Sparse Distributed Memory	57
3.3	Comparison with a traditional computer	58
3.4	Discussion	59
4	The Hopfield network	63
4.1	The traditional Hopfield network	64
4.1.1	Belief propagation in traditional Hopfield networks	66
4.1.2	Coordinate descent on the joint probability	68
4.1.3	Maximum entropy decoding	69
4.1.4	Uncertainty in the cue	71
4.1.5	Results	72
4.1.6	Related Linear networks	72
4.2	Improving on pairwise encoding	76
4.2.1	The neurally-inspired Hopfield network	76
4.2.2	The Neurally-inspired Counting Bloom filter	79
4.3	Conclusions	81
5	Associative Memory from Error-Correcting Codes	83
5.1	Introduction to Error-Correcting Codes	84
5.2	Simple Error-Correcting Codes	85
5.2.1	Encoding	87
5.2.2	Calculation of Messages	89
5.2.3	Theoretical Insight for the parity code	89
5.2.4	Performance	92
5.2.5	Efficiency	94
5.3	Cued error-correction	95

CONTENTS

5.3.1	Minimum connectivity	96
5.3.2	Results	98
5.4	Population codes	99
5.4.1	Allocation	101
5.5	Conclusion	106
6	Associative Memory in Continuous Time	109
6.1	Related work	112
6.1.1	Synfire networks	113
6.1.2	Polychronization	114
6.1.3	Concurrent Recall Network	115
6.1.4	Self-organising recurrent networks	115
6.2	Description of the model	116
6.2.1	Detecting successful retrieval	122
6.3	Measuring Information	124
6.3.1	Information of whole patterns	124
6.3.2	Information of individual spike times	125
6.4	Stored versus recalled patterns	126
6.5	Synaptic strength and transmission delays	128
6.6	Information Retrieval and Pattern Size	131
6.6.1	Maximum I_2 recall performance	134
6.6.2	Pattern Size	135
6.6.3	Linear Capacity	136
6.7	Discussion	137
6.7.1	Future work	138
6.7.2	Conclusions	140
7	Conclusions	143
7.1	Overview	144
7.2	Discussion	146
A	Commonly Used Symbols:	149

CONTENTS

B Commonly Used Symbols:

Associative Memory in Continuous Time **151**

References **162**

CHAPTER 1

Introduction

The amazing flexibility of the brain stands in stark contrast to the rigidity of today's computers. While congenitally blind people are able to re-use their visual cortex for other sensory processing (Pascual-Leone *et al.*, 2005), a single faulty transistor out of the billions in a computer chip can render the entire chip useless. This dissertation considers performing useful computation in an efficient manner that is robust to both the failure of hardware and the re-allocation of hardware to other tasks (Wermter *et al.*, 2001). While biological brains have large amounts of structure at the gross anatomical level, it seems unlikely that individual neurons are able to specify exactly which cells they connect to. Yet biological cells perform their computation in spite of this randomness. Whilst there is certainly some order in the synaptic connectivity, it is impossible for every synaptic connection in the brain to be explicitly specified in the genetic code. This suggests that there is a short description (Hinton & van Camp, 1993) for the wiring of biological brains, and we will explore methods which can be described in their entirety (including their wiring) using a small number of parameters. The minimum description length (Rissanen, 1986) of a model is related to the accuracy required to describe a model. The models we discuss here require very little accuracy as their random connections do not have to be described at all. It seems plausible that devices with short description length would be easy to evolve.

The human neocortex has rapidly increased in size over the past few million years (Rakic, 1995) which suggests that evolution has found a computational structure whose computational power can be scaled up by simply adding more of the same (Mountcastle, 1997). This dissertation will deal with similar structures which increase their computational power by increasing the number of computational units available. Strictly speaking, some of the designs covered in this dissertation have parameters which need

1. INTRODUCTION

to change logarithmically as more computational units are added, but this is slow compared to the growth in computational units.

We hope that this work will provide some insight into the nature of the computation performed by the brain. One can better understand the brain either by building detailed models of biological neurons and observing the behaviour of the network (Markram, 2006), or by considering the computational tasks they perform and deriving a useful model. It seems plausible that the set of computational tasks is smaller and easier to explore than the set of possible interactions in detailed neural network models. The latter is complicated by the abundance of different neural models and there is considerable uncertainty in how information is coded by biological neurons. For these reasons this dissertation examines principled solutions to computational tasks, while accepting that the solutions obtained should also achieve certain design goals that are consistent with biological neurons. For this dissertation two of the most important design goals are:

Inability to specify exact connectivity: At the gross anatomical level there exists detailed structure in neural connectivity. As an example, the neocortex has distinct layers of neurons which send dendrites out to other distinct layers (Bear *et al.*, 2007). However at the level of individual cells we believe that it is impossible to specify precisely which other cells should be connected. It is at this level in vertebrates that connections are somewhat random.

Robustness to hardware failure: Neurons are not only very noisy pieces of hardware, they are also liable to die at any time. However we do not expect this failure to catastrophically affect the computations carried out by other neurons. This implies that we need to use distributed representations with no single points of failure (Hinton *et al.*, 1984).

Using these design goals we will seek the best principled solution for our computational task. While we might draw our inspiration from biology, we do not feel constrained by it. If we find a solution that is not biologically plausible then we can examine the solution to find other important constraints in the design of biological neurons, or we will have weak evidence that biological neurons do not solve our particular computational task.

1.1 Associative memory

In this dissertation we restrict ourselves to the computational task of associative memory. The associative memory task can be described as follows:

Given a set of items $\{x_1, x_2, \dots, x_R\}$, create a representation z which contains information about all the items. When presented with a corrupted version (or cue) of the r^{th} pattern c_r , use the information stored in z to “clean up” c_r and return x_r .

As an example of how humans effortlessly solve the associative memory task, consider how many different pieces of music one has heard in one’s lifetime. The music can take many different forms e.g. songs, movie themes, and advertising jingles. Sometimes the piece is heard only once or twice and the music is forgotten. Certain pieces of music might be heard many times and one becomes an expert at recognising them. It usually only takes a few notes to trigger the recall of the rest of the song, even if the notes are played incorrectly by a practising musician. The music can also become powerfully linked to other events in our lives so that hearing the music will trigger the recollection of a particular period in our lives or a particular person.

We are interested in solving the associative memory task by creating a brain-like device. Our computations will be performed using many identical computational units. Each computational unit is able to perform relatively simple computations and will have a small amount of storage associated with it (e.g. a single bit, or a single integer). Individually these units could not solve the task, yet together many simple identical units can perform the computation through communication with each other. Associative memory has to be able to store and recall patterns, and we assume that the units are able to change from one stage to another in a globally coordinated manner.

Each associative memory solution we propose is principled, but has different properties; e.g. it operates in continuous time, or it gradually forgets old patterns. For most of this thesis we will consider storing random binary vectors, although we will also consider spatio-temporal neural spike patterns in chapter 6. Some associative memories have binary storage with which to store representations of the patterns, others have integer storage. However all use principled statistical inference to perform recall, and use message-passing to infer the marginal probability of each bit.

1. INTRODUCTION

We will perform the inference using high-precision messages, yet in a real neural system such precise messages would be very expensive, or even impossible. However we believe that with clever encoding strategies it is possible to achieve a similar level of performance using low-precision messages. Such low-precision messages have already been found for error-correcting codes (Richardson & Urbanke, 2001), so there is already a large literature to pursue further, but we will not cover it in this dissertation.

For the chapters in which we consider binary vectors as the items of interest, we will consider a simplified form of the noisy cue. We assume that our cue will consist of a binary vector, in which we know that some fixed percentage of the bits are flipped. With the probabilistic framework we use here we could consider much more general situations. As an example we could consider the case whereby half the cue is known with complete certainty, while the other half is completely uncertain (this would correspond to storing a mapping between keys and values). However, by restricting ourselves to the simpler uncertainty model we are able to compare our methods with non-probabilistic methods, e.g. the Hopfield network in chapter 4.

1.2 Recall as inference

Associative memory can be done in a principled probabilistic framework. Given some stored functions of previous inputs z and an initial cue x , we can use Bayes' rule to calculate the posterior belief (Sommer & Dayan, 1998):

$$\Pr(x | z) = \frac{\Pr(z | x) \cdot \Pr(x)}{\Pr(z)}. \quad (1.1)$$

The initial cue provides a prior belief over x , and to avoid confusion we will use c to refer to the prior, while we will use x to refer to our final estimate of the pattern. Some functions will have a better capacity, and some will be better suited for distributed message-passing. These functions are more relevant for constructing neurally-inspired associative memories.

Principled statistical inference allows one to trivially consider different types of faulty hardware. As long as one can characterise the fault in a probabilistic manner, then one can adjust the evidence from observing z .

At the end of our computations we are then left with a posterior probability for each of the bits in x : $\Pr(x | z)$. We will convert this posterior back into a bit string as

this allows us to compare our methods against non-probabilistic methods (such as the Hopfield network in chapter 4). If we were using our associative memory as part of a much larger system then we would want to ideally convey the certainties associated with each bit as well.

1.3 Capacity

If we are trying to create an associative memory in a noisy environment where individual components can fail at any time, using probabilistic reasoning to perform recall, then we must accept the possibility of errors in our final recall. We must then define the performance of a noisy recall. This dissertation will argue that the natural measure can be taken from information theoretic results on the capacity of a binary symmetric channel.

This new measure of capacity is more complex than previous measures of associative memory capacity. Associative memories which are based on ad-hoc rules of recall tend to have catastrophic failure when they store too many patterns. The point of catastrophic failure can then be defined as the capacity of the associative memory. Principled inference avoids such catastrophic failure as the recall performance gracefully declines to the original cue, and hence motivates the need for a new measure.

An associative memory is only useful when it is able to *improve* the initial cue. For very noisy cues there might not be enough information to identify the correct pattern, while noise-free cues would not require an associative memory in the first place. As such, when evaluating the capacity of an associative memory we will examine the performance under different levels of noise.

1.4 Common themes in our solutions

We discuss many architectures that differ in the implementation details and the properties of the memories but there are still themes common to them.

Random connectivity: A major theme of this dissertation is that principled computation can be carried out efficiently in a distributed manner, even in the face of random connectivity. If we are interested in building brain-like devices then ideally we should

1. INTRODUCTION

be using the simplest components possible. If a device works (with high probability) using random connectivity then that device has a short description length (Hinton & van Camp, 1993); one need not describe the exact connections, but can specify that the connections are wired up randomly. This dissertation will provide evidence that random connectivity still produces near-optimal results.

Distributed representations: Using random connectivity and independent simple components, leads one very naturally to consider a distributed representation (Plate, 2003) for the information stored in z . Using a distributed representation ensures that the loss of any particular piece of hardware is not catastrophic.

Hashing: A distributed representation which uses functions of randomly selected bits can also be viewed as a hashed representation. Hashing (Mitzenmacher & Upfal, 2005) is a great idea when building brain-like devices as one does not need to worry about complicated addressing schemes, instead one creates a distributed hash of the item and takes care to ensure that there is a low probability of a collision occurring, or learning similarities in the data so that similar items get hashed to similar representations (Salakhutdinov & Hinton, 2009).

Correctness as a resource: We accept that our computations may end with a small number of bit errors. We also find in this noisy environment that allocating more computational resources lowers the long-term error rate. This dissertation views correctness as a resource that can be traded (just like the space and time requirements of many algorithms). By accepting noisy results we hope to find simple algorithms that achieve acceptable results fast. We also hope to do away with infrequent algorithmic complications that occur in other well-known algorithms, e.g. collision resolution in hashing (Goodrich & Tamassia, 2009).

1.5 Overview of the dissertation

In chapter 2.1 we start by considering familiarity detection; a task intimately related to associative memory, yet much simpler. To solve the familiarity detection task we must be able to answer whether or not a pattern has been seen before (rather than cleaning up a noisy cue). We then build on our solution for the familiarity task and turn the

solution into an efficient associative memory in chapter 3. In the same chapter we also explore a variant of the associative memory which gradually forgets old patterns.

We find that principled statistical inference avoids the catastrophic failure common to many associative memories. However an associative memory is typically defined by its point of catastrophic failure. This motivates the need for a new measure of capacity. However our capacity results are meaningless without a well-known benchmark to compare to. In chapter 4 we compare against the best-known network, the Hopfield network and explore several principled methods of recall in a Hopfield network.

Cleaning up a cue can also be thought of as error correction, and in chapter 5 we show an associative memory that is based on simple error-correcting codes. This associative memory is particularly efficient for strong cues which require very little clean-up, but the architecture requires more complicated hardware than previous associative memories.

We finally move onto a network that operates in continuous time and whose elements are most like biological neurons in chapter 6. We show that this network can fill in missing spikes, align noisy spikes and, excitingly, it can recall many patterns simultaneously. Finally in chapter 7 we summarise all of our findings and conclusions.

In summary then, we are interested in performing computation in a robust distributed manner, using random connectivity, yet still using principled statistical inference to achieve the computation. In this thesis we restrict ourselves to the associative memory task and explore the properties of several solutions, but believe our approach could be applied to many other types of computational tasks.

1. INTRODUCTION

CHAPTER 2

The neurally-inspired Bloom filter

In later chapters we will tackle the associative task, but we first begin with a similar, simpler task: familiarity detection. There is already some literature on familiarity detection using neural networks, which we cover in section 2.3. We instead build on a solution from computer science that is known to be nearly optimal; the Bloom filter (Broder & Mitzenmacher, 2004).

A *Bloom filter* is a data structure that provides a probabilistic test of whether a new item belongs to a set (Bloom, 1970). We are interested in representing a set of items. As an example, consider the set of “people I have met”. The representation will need to perform two tasks to be useful. We need to be able to query the set to determine if we have met a person before. The representation also needs to be able to add people as one meets new people. For our purposes a person can be represented by the presence/absence of various features, which can be directly mapped to a binary vector (we will assume these features are independent).

A Bloom filter has a small (but controllable) probability of a false positive. In our example this means there is a small probability of the Bloom filter reporting that we have met a person when in fact we have not. Accepting a small probability of error allows far less memory to be used (compared with direct storage of all the items) as Bloom filters can store *hashes* of the items.

Our Bloom filter is created from many identical parts. Each part will have a different random boolean function and a single bit of storage associated with it. In this approach, storage and processing are closely related. Our network is similar to Willshaw nets (Willshaw *et al.*, 1969) and Sigma-Pi nets (Plate, 2000) that both store items using simple logical functions. Their similarities will be better covered in section 2.3 after the standard Bloom filter and its proposed modifications have been fully described.

2. THE NEURALLY-INSPIRED BLOOM FILTER

Our approach is *neurally-inspired*: Biological neural networks are able to perform useful computation using (seemingly) randomly-connected simple hardware elements. They are also exceptionally robust and can lose many neurons before losing functionality. Analogously, our approach is characterised by storing simple random functions of the input and performing distributed statistical inference on the stored values. The individual parts of our Bloom filter work in parallel, independently of each other. Since the parts are independent, the inference is able to continue even if some of the parts fail.

In this section we first describe related work. We then describe the standard Bloom filter and derive the theoretical behaviour before moving on to the neurally-inspired version. A good introduction to Bloom filters is given in Mitzenmacher & Upfal (2005), and a shorter introduction is given in the next section.

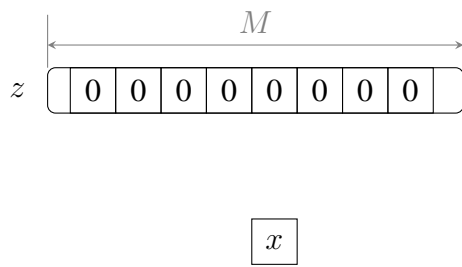
2.1 The Bloom filter

A Bloom filter (as shown in figure 2.1) consists of a vector z (containing M bits), and K hash functions $\{h_1, \dots, h_K\}$. The hash functions map the set of all possible elements to be stored onto indices for z . If the hash functions are good, then all indices are equally likely to be used.

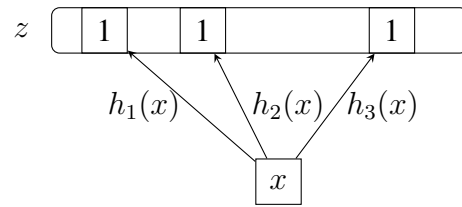
To represent the set we initialise all elements of z to zero. Each time we add an element x we calculate the indices $\{h_1(x), \dots, h_K(x)\}$, and turn these bits on. If a bit has already been turned on, then it remains on.

To test whether or not \hat{x} is an element of the set we calculate all K indices $\{h_k(\hat{x})\}$; if any of the locations in z contains 0 then we are certain that the item is not in the set. If all of the indices are 1, then \hat{x} is probably part of the set, although it is possible that the bits were turned on by storing other elements. To estimate the probability of a false positive (for an element that is not in the set) we let p denote the proportion of 1's in z . For perfect hash functions all indices are equally likely and the probability that all of the $h_1 \dots h_K$ locations are one is:

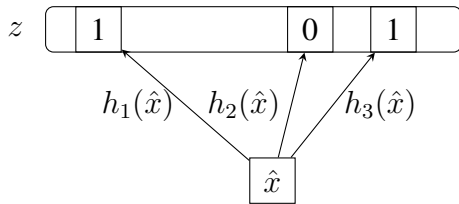
$$\Pr(\text{false positive}) = p^K. \quad (2.1)$$



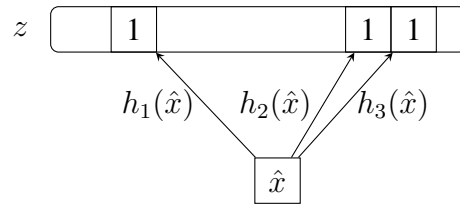
(a) Initially all elements in z are set to 0.



(b) To add x to the set we calculate the hash functions $h(x)$ which provide indices into z . All bits at those locations are set to 1.



(c) To test \hat{x} for membership, we examine the bits pointed to, $\{h(\hat{x})\}$; if *any* of the bits pointed to are zero then \hat{x} is *not* a member of the set.



(d) If all the bits pointed to are 1, then the Bloom filter reports that \hat{x} is in the set, although there is a small probability that it is not and all these 1's have actually been set by storing other elements.

Figure 2.1: Overview of a traditional Bloom filter.

2. THE NEURALLY-INSPIRED BLOOM FILTER

If we are storing R items and they each have K hashed indices, then we can estimate the proportion p of 1's in z :

$$p = 1 - \left(1 - \frac{1}{M}\right)^{KR} \approx 1 - e^{-KR/M}, \quad (2.2)$$

where M is the number of bits in Z . If we are given the size of our storage, M , the number of items to store, R , and we would like to minimize the probability of a false positive by tuning the number of hash functions then we can imagine two competing forces. The first is that as we increase the number of hash functions we have more chance to encounter a zero when a non-member is presented. However as we increase the number of hash functions we also increase the proportion of 1's in z (Mitzenmacher & Upfal, 2005). Differentiating $\ln \Pr(\text{false positive})$:

$$\frac{\partial \ln p^K}{\partial K} = \ln(1 - e^{-KR/M}) + \frac{KR}{M} \frac{e^{-KR/M}}{1 - e^{-KR/M}}. \quad (2.3)$$

As one might guess (guided by information theory) $p = 0.5$ produces $K = \frac{M}{R} \ln 2$ and indeed this is the only zero of equation 2.3. Thus the false positive probability is minimized by making z look like a completely random bit string. Of course the number of hash functions used, K , has to be an integer so in practice the results might be slightly higher than predicted.

It can be tedious to find good hash functions even for the typically small number of hash functions (e.g. $K = 20$), Kirsch & Mitzenmacher (2005) have shown that linear combinations of only 2 hash functions can still create a large number of good hash functions. This is the approach we took to achieve the results given later in this chapter.

A Bloom filter can also be thought of as performing complicated bit-wise calculations to carry out its computations. In figure 2.2 we show a Bloom filter that calculates the same hash functions for each bit in z and tests whether any of those hash functions would set the bit. To literally program a Bloom filter in this way would be wasteful, as the same computation is being repeated for each bit. However this view is suggestive of a generalization to the Bloom filter. Rather than calculate a complicated hash function for each bit, we will instead calculate a simple random boolean function of a subset of the bits in x . We will choose different random functions and different subsets for each bit in z (but once we have chosen a function and a subset we will use it for all

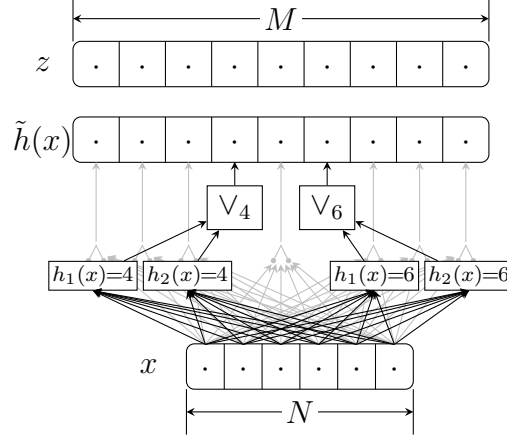


Figure 2.2: An alternative view of the Bloom filter computation. For bit m we calculate the K hash functions (here $K = 2$) and test whether any of the hash functions are equal to m by taking the logical OR (\vee). This is an impractical way to perform the necessary Bloom filter computations as the hash functions will be recalculated for each bit. However it suggests that we might be able to obtain similar performance through a simpler bit-wise computation.

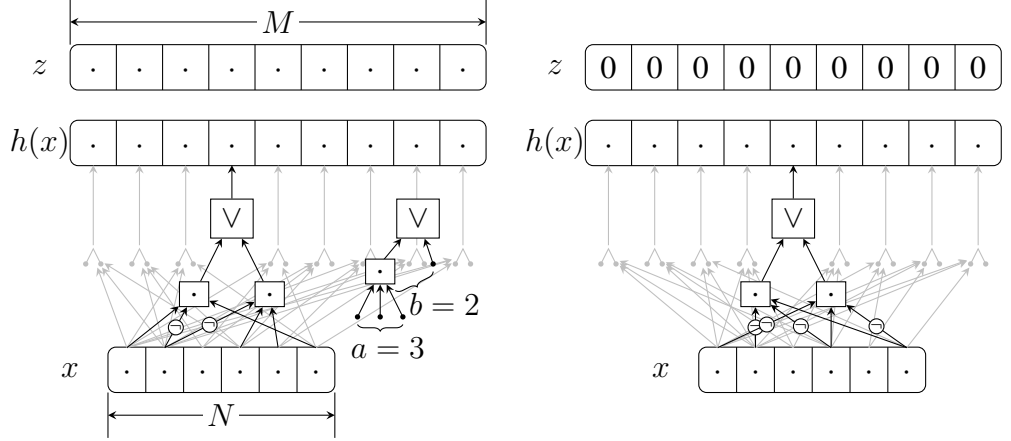
the storage and queries we require). We call this a neurally-inspired Bloom filter and cover it in detail in the next section.

2.2 Neurally-inspired Modifications

A standard Bloom filter can be turned into a neurally-inspired Bloom filter (as shown in 2.3) by using a simple boolean hash function for each bit in place of the set of K hash functions (as shown in figure 2.2). We denote these functions h_m to indicate that they serve a similar function to the original Bloom filter hashes, and we use the m to indicate that there are no longer K of them, there are now M – one for each bit in z .

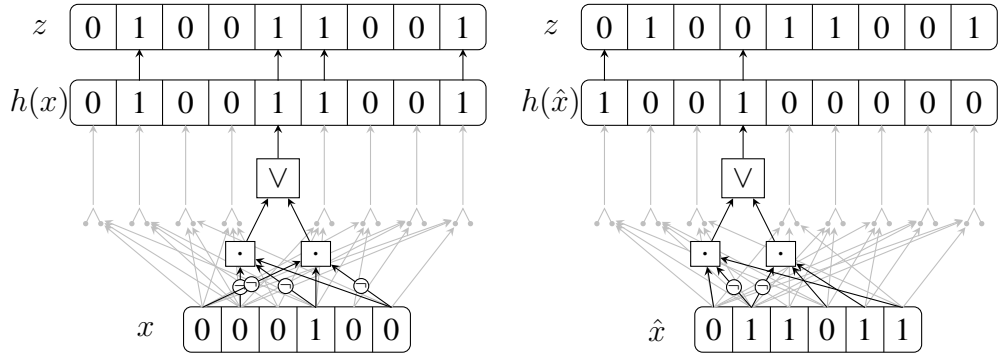
For our boolean function, we assume x is a binary vector and work with individual bits in x . Our function “OR”s lots of randomly negated bits that have been “AND”ed together – also known as a sigma-pi function (Plate, 2000). While it is silly to name the function after the Greek letters (sigma for sum and pi for product) the term “sum-product” although more descriptive conflicts with the sum-product algorithm which

2. THE NEURALLY-INSPIRED BLOOM FILTER



(a) Each bit has a function which “AND”s many variables and “OR”s the result.

(b) We initialise all elements in z to zero.



(c) Adding an item.

(d) Testing for membership.

Figure 2.3: In a neurally-inspired Bloom filter we associate a boolean hash function with each bit in z . To store x in the set each hash function calculates its output for x and if true then the associated bit is set to one. In (d) we can be sure that \hat{x} has never been stored before, as two elements of $h(\hat{x})$ return true, but the associated bits in z are false.

2.2 Neurally-inspired Modifications

we will use in subsequent chapters. An example of the functions we consider is:

$$h_m(x) = (x_1)(\neg x_6)(x_3) \vee (\neg x_3)(x_9)(\neg x_2) \vee (\neg x_5)(x_6)(x_4). \quad (2.4)$$

The variables that are “AND”ed together are chosen at random, and varied for each bit of storage. Each hash function has the same form and once a hash function has been chosen for a bit of storage then it is fixed for the remaining operations of the Bloom filter. These functions have the advantage that they are simple to evaluate, and they have a tunable probability of returning true. To increase or decrease this probability of returning true, one can increase or decrease the number of terms we “OR” together. If we want finer gradations in the changes of the probability of returning true then we can increase the number of “AND”s in each term.

We also include random negations of the bits (e.g. $\neg x_6$). This removes the dependence on x having a certain proportion of bits on to function correctly. As an example of how the network would fail without the random negations consider storing $x^\top = [1, 1, \dots, 1]$. All boolean functions would return true and z would be saturated.

To initialise the network we set the storage z to the all-zero vector. We store information by presenting the network with the input and whichever boolean functions return true, the associated storage bit z_m is turned on.

To query whether an \hat{x} has been stored, we test whether any of the boolean functions return true, yet the corresponding bit in z is false. If this occurs then we can be sure that \hat{x} has never been stored. Otherwise we assume that \hat{x} has been stored, but there is a small probability that we are incorrect, and the storage bits have been turned on by other data.

Given the number R of items to store, we can find the form of the boolean functions that minimize the false-positive probability. If we denote by p the probability that a single function returns true then the probability p^* of a bit being true after all R memories have been stored is:

$$p^* = 1 - (1 - p)^R. \quad (2.5)$$

We can estimate the false positive probability, for a given value of p , by estimating the number of checks that are performed (on average there are Mp of them), and observing that each one will pass with probability p^* . However, items that activate fewer than

2. THE NEURALLY-INSPIRED BLOOM FILTER

Mp checks will be more likely to fail. We denote by k the number of checks and in the absence of knowledge of the network structure h or the input x we note that the distribution of k is well-approximated by a Poisson distribution. The probability of failure can be obtained by marginalising over k :

$$\begin{aligned} \Pr(\text{failure}) &= \sum_k \Pr(k) \cdot \Pr(\text{failure} | k) \\ &= \sum_k \left(\frac{(Mp)^k e^{-Mp}}{k!} \right) \cdot p^{*k} \\ &= e^{-Mp(1-p)^R}, \end{aligned} \tag{2.6}$$

which has a minimum when:

$$p = \frac{1}{R+1} \tag{2.7}$$

again and results in $p^* \approx 1 - e^{-1}$ of the bits being on after all memories have been stored. Therefore the storage does not look like a random bit string after all the patterns have been stored. If we examine figure 2.4 we see that the performance when the saturation is 0.5 is worse than the predicted optimal saturation. This difference occurs because of the noise in the number of checks the neurally-inspired Bloom filter makes. It is now optimal to make slightly more checks on average than the original Bloom filter, even if this means more bits are turned on in z .

If p is the target probability, we have to choose how many terms to “AND” together and then how many of such terms to “OR”. Ideally we would like:

$$p = 1 - (1 - 2^{-a})^b, \tag{2.8}$$

where a is the number of “AND”s and b is the number of “OR”s. For large R and using equation 2.7, this implies the approximate relation:

$$b \approx \frac{2^a}{R+1}. \tag{2.9}$$

Contrasting this result with a traditional Bloom filter it is interesting to note that there is no dependence on the size of the storage M . The neurally-inspired Bloom filter automatically scales with any amount of storage, whereas optimal performance in a traditional Bloom filter dictates that the number of hash functions must be changed if the size of the storage changes.

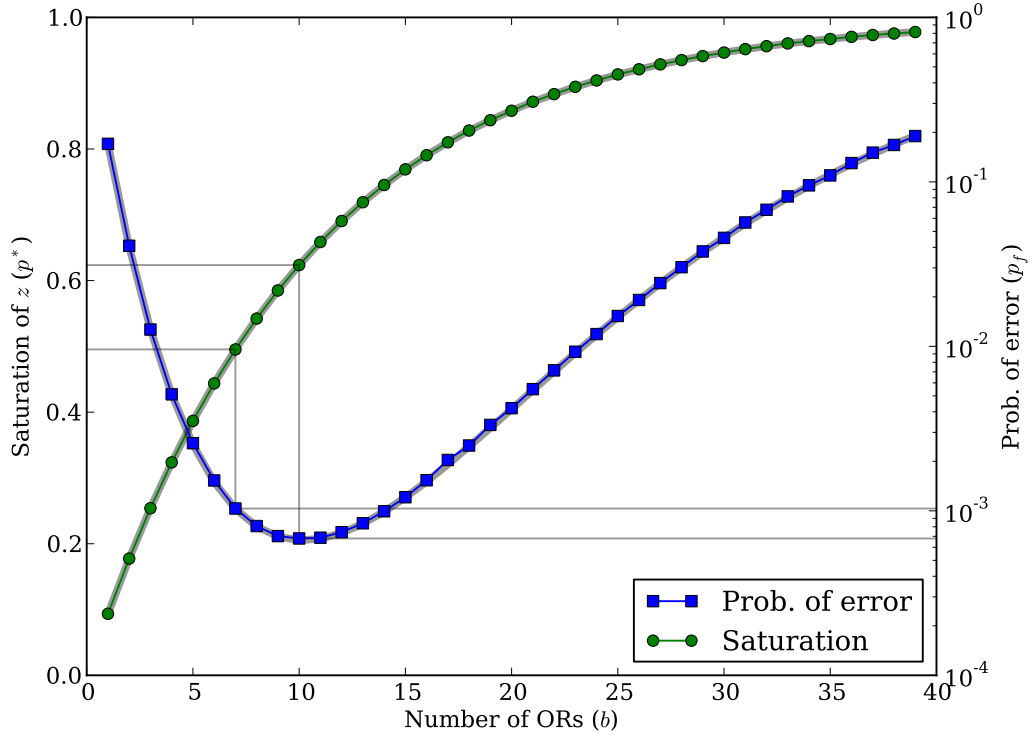


Figure 2.4: The performance of the neurally-inspired Bloom filter for different boolean hash function parameters. By varying the number of terms that are “OR”ed together we can change the final saturation in z , which affects the probability of a false-positive. This figure was created by storing $R = 100$ items in $M = 2000$ bits of information, with $a = 10$ distinct variables “AND”ed together. The expected theoretical performance for both curves are plotted with a grey line and match the empirical results closely. We also show that the error rate for the saturation $p^* = 0.5$ is non-optimal and the optimum instead occurs when $p^* = 1 - e^{-1}$.

2. THE NEURALLY-INSPIRED BLOOM FILTER

To store R patterns in a traditional Bloom filter with a probability of false positive failure no greater than p_f requires a storage size of at least:

$$M = R \frac{-\ln p_f}{\ln^2 2}, \quad (2.10)$$

while a neurally-inspired Bloom filter would require a storage size:

$$M = R(-\ln p_f)e. \quad (2.11)$$

We see that for an equal probability of bit failure the neurally-inspired version requires a constant factor ($e^{-1} \ln^2 2 \approx 1.31$) more storage than the Bloom filter. This is the price one pays for randomly choosing bits in parallel, independently of all other bits. If one were able to ensure that exactly Mp functions were activated every time, then the optimal saturation would again be 0.5 and the performance would be identical to the traditional Bloom filter. It is impressive that the required space for a neurally-inspired Bloom filter is within a constant multiple of the optimal given how simple its boolean hash functions are.

2.2.1 Storage flexibility

If there are many neurally-inspired Bloom filters operating in an environment, then one can imagine contexts where a new set is required to be represented, and one can simply take storage from other neurally-inspired Bloom filters. The other filters will *not* have a higher probability of error if we only take units where the corresponding bit has been set and they will still be able to function as before. (Implicitly we are lowering the error rate by using the extra information contained in our choice of which units to remove.) Even if we are unable to choose which hardware to re-allocate, equation 2.9 tells us that the reduced storage will still be operating optimally (as the parameters of the ideal boolean hash functions do not depend on the storage size M).

While we can easily remove storage from a Bloom filter we are unable to add storage to an already-created Bloom filter without running the risk of introducing false negatives, i.e. reporting that an element is not in the set when it actually is. This limits our flexibility as ideally we would like to be able to add and remove storage as we see fit. There are two approaches to lessen this limitation. Firstly one could

use a conservative strategy of over-allocating storage when creating a new neurally-inspired Bloom filter and then gradually removing storage as one is certain that there really is excess storage. Secondly scalable Bloom filters do exist (Almeida *et al.*, 2007) which allow one to add storage to a Bloom filter, but they require the coordination of several traditional Bloom filters. To allocate more storage to a scalable Bloom filter one creates a second Bloom filter and stores new items only in the new Bloom filter. An item is deemed to have been stored in the scalable Bloom filter if it is in any of the individual Bloom filters. As such the error rate can never go down with the addition of more storage, but if one is careful one can ensure the total error rate is bounded by some maximum, regardless of the number of items stored. Requiring the coordination of several Bloom filters runs counter to the neurally-inspired design requirement of achieving computation through low-level, distributed, local actions and we did not explore this option any further.

2.3 Related Work

This section describes the Willshaw network (Willshaw *et al.*, 1969) and the Sigma-Pi network (Plate, 2000). Both are relevant to the neurally-inspired Bloom filter, as they store patterns by using the results of simple binary functions of the patterns. However the methods create heteroassociative memories which associate an input pattern x together with an output pattern y . For a query these methods assume a noiseless input and then infer the output. It is straightforward to turn these associative memories into familiarity detectors. We also cover work which uses a Hopfield network for familiarity detection and other related Bloom filter work.

2.3.1 The Willshaw network

The Willshaw network associates a sparse binary input together with a sparse binary output in a binary matrix which describes potential all-to-all connectivity between all possible pairs of inputs and outputs. Initially all the associations between the input and output vectors are off. When a (x, y) pair is stored then the connections between active inputs and active outputs are turned on (see figure 2.5). During recall time just an x is presented and the task is to use the synaptic connectivity to estimate what y should be.

2. THE NEURALLY-INSPIRED BLOOM FILTER

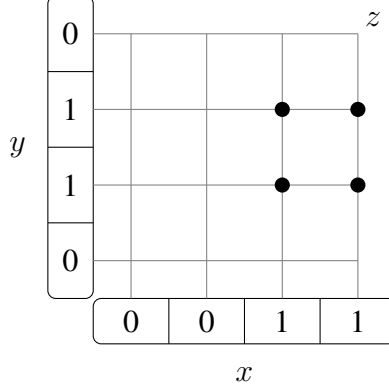


Figure 2.5: The Willshaw associative network stores pairs of patterns $\{(x, y)\}$. To store a pattern z_{ij} is updated according to the following rule: $z_{ij} = z_{ij} \vee x_i y_j$. The Willshaw network is designed to store sparse binary patterns. Given a correct x we can then use z to recall the associated y .

While many recall rules have been proposed (Graham & Willshaw, 1995, 1996, 1997) the most relevant performs proper statistical inference (Sommer & Dayan, 1998) to retrieve the maximum likelihood estimate.

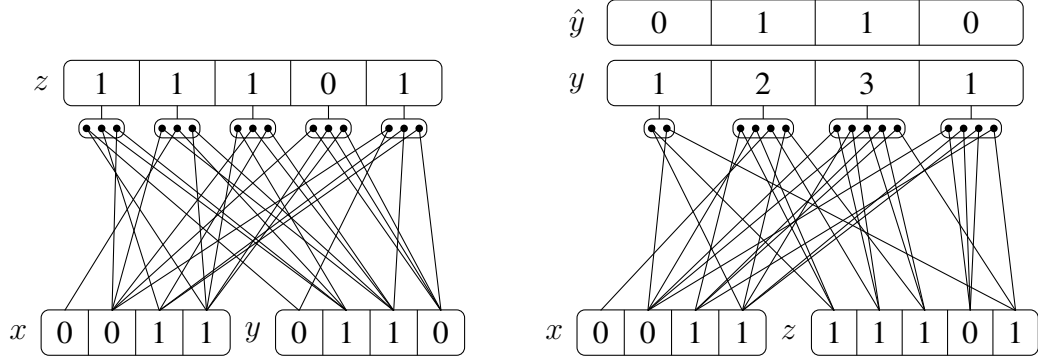
A Willshaw network can represent a set of items in a straightforward manner. If the network is queried with a pair of binary vectors (\hat{x}, \hat{y}) and one can find $z_{ij} = 0$ yet $x_i = 1$ and $y_j = 1$ then one knows that the pattern has not been stored before. This assumes that the synaptic connectivity is completely reliable, but does allow a simple estimate of the capacity. As a further simplification we will consider the case that the dimensionality of x and y are equal (N). Since Willshaw networks are efficient when the patterns are sparse ($\sum_i x_i = \log_2 N$), if we store approximately:

$$R = \frac{N}{\log_2 N} \quad (2.12)$$

patterns, then the probability of a bit in z being on is 50%, and the probability of getting a false positive for a randomly-chosen pair of binary vectors x' and y' is:

$$\Pr(\text{failure}) = 2^{-(\log_2 N)^2} = N^{-\log_2 N}. \quad (2.13)$$

This gives the rate at which false positives decrease for increasing pattern sizes. Alternatively we can think about a maximum probability of error p_f and store more patterns



(a) The Sigma-Pi network encodes an association between x and y by calculating the sum of random products and storing the results in z .

(b) Given the inverse mapping together with the vectors x and z the network can compute a sum of products to estimate y .

Figure 2.6: The Sigma-Pi associative network.

as the pattern sizes increase:

$$R \approx \frac{-N}{\log_2 N} \cdot \log \left(1 - p_f \left(\frac{1}{\log_2 N} \right)^2 \right). \quad (2.14)$$

Notice that a Willshaw network is not able to store dense binary vectors very well. After $R = 20$ patterns have been stored then the probability that a particular connection has not been turned on is on the order of one in a million. This is a very uninformative state and suggests that one of the key insights of the neurally-inspired Bloom filter is to use functions which are able to create sparse representations (for an arbitrary level of sparsity).

2.3.2 The Sigma-Pi associative network

The Sigma-Pi network (Plate, 2000) creates a heteroassociative network out of Sigma-Pi functions. The pairs of patterns are sparse (at a similar sparsity to the Willshaw network). The Sigma-Pi functions consist of randomly chosen terms which “AND” a bit from x together with a bit from y (see figure 2.6(a)).

The work is interesting as Plate shows that recall can also be achieved with Sigma-Pi functions. If a term in the Sigma-Pi function for z_k “AND”s together a bit in x_i and

2. THE NEURALLY-INSPIRED BLOOM FILTER

y_j , then one should create a term in the decoding function for bit y_j which “AND”s together bits x_i and z_k (compare figures 2.6(a) and 2.6(b)). While this decoding rule is not Bayesian (preliminary experiments not reported here suggest four times as much capacity with Bayesian decoding) it is conceptually satisfying to have both the encoding and decoding performed in the same manner. The inverse mapping can also be learnt in a relatively short time frame by observing the effect of random input patterns (x, y) on z .

2.3.3 Familiarity detection with Hopfield networks

Particularly relevant work to this dissertation is Bogacz *et al.* (1999). Although they describe the task as familiarity discrimination, they essentially represent a set of previously seen elements using a Hopfield network. (This thesis will cover the Hopfield network in greater detail in chapter 4.) After carefully examining the statistics for the number of bits that flip when performing recall of a random element they then determine a threshold for declaring an element unseen or not. They define “reliable behaviour” as “better than a 1% error rate for a random, unseen element”. Their method can generate both false positive and false negatives. From equation 2.11 our Bloom filter requires 13 bits per pattern for a 1% error rate and gives only false positive errors. We can phrase this result in their terminology. If we scale our storage size M similarly to a Hopfield network with increasing input size N , ($M = N(N - 1)/2$) then we can store:

$$R = \frac{N(N - 1)}{2 \times 13} \quad (2.15)$$

patterns, which to leading order is $R = 0.038N^2$ and compares favourably to their reported results of $0.023N^2$. It is worth emphasizing that we achieve superior performance using only binary storage while Hopfield networks use integers. We can also invert the process to find that a Hopfield network needs approximately 22 integers per pattern for a 1% recognition error rate.

Greve *et al.* (2009) also consider creating a recognition memory from a Hopfield network, however they consider the more general case where the binary patterns to be stored can have a bias (i.e. there are significantly more 0’s than 1’s). When storing biased patterns some conditions are more likely than others and they allow the different conditions to affect z differently. Allowing the effects to vary, they then derive a

learning rule which maximises the signal-to-noise ratio for familiarity detection. It is important to note that Greve *et al.* (2009) define their threshold in terms of the value of the energy, while Bogacz *et al.* (1999) define their threshold in terms of the change in energy after the first step of recall has been performed. It appears that the threshold in terms of energy performs better, but still not as well as the neurally-inspired Bloom filter.

2.3.3.1 Palimpsest familiarity detection

It is possible to create representations of a set which gradually forget old items. These are known as palimpsest networks and are covered in greater detail for associative memory in section 3.2.

Barrett & Van Rossum (2008) examines the capacity of a single neuron to recognise previously-seen patterns. The neuron has a synapse (unit of storage) for every bit in an input pattern. Each synapse can only take a discrete number of possible states. A pattern is stored by updating the values in a stochastic manner. If an input bit is on then the associated binary synapse will be switched on with probability p . If the input bit is off then the binary synapse will be switched off with probability q . As many patterns are stored then older patterns are overwritten by newer patterns and is therefore a palimpsest. Barrett & Van Rossum (2008) is interested in the capacity of the neuron for familiarity detection, which the previous chapter has shown to be intimately related with associative memory. They are able to calculate the information contained per synapse and find that the information per synapse in a network of N neurons falls off as $O(\frac{1}{\sqrt{N}})$. Our findings suggest that it is possible to ensure the information per synapse (unit of storage) is constant, which suggests that their procedure is sub-optimal. It appears this change in performance is due to an un-optimised level of sparsity in the patterns which are stored. Our work suggests that the optimal sparsity will depend on the number of patterns stored. This needs to be interpreted with some care as the settings for each are different.

Barrett & Van Rossum (2008) also explore the more general setting of multi-state synapses and find the optimal equilibrium distribution which is of a particularly interesting form. The optimal distribution places more probability mass on the minimum

2. THE NEURALLY-INSPIRED BLOOM FILTER

and maximum settings of the synapse with correspondingly less mass on the intermediate states. This suggests possible future work to find better encoding functions with non-binary synapses which would achieve this equilibrium distribution.

2.3.4 Related Bloom filters

A counting Bloom filter (Broder & Mitzenmacher, 2004) is a variant which allows an item to be removed from the set and could also be turned into neurally-inspired variant, by allowing the storage of an integer rather than a single bit. A neurally-inspired counting Bloom filter can be constructed in a straightforward manner. In the next chapter we will turn the neurally-inspired Bloom filter into an associative memory, and in section 4.2.2 on page 79 we briefly examine the performance of our counting Bloom filter as an associative memory.

The most similar variant of a traditional Bloom filter to our neurally-inspired version is the partitioned Bloom filter. In Kirsch & Mitzenmacher (2008) a brief description of the partitioned Bloom filter is given. z is partitioned into several shorter vectors $z_1 \dots, z_K$ each with its own hash function. Each hash function is restricted to only index into its associated vector. This allows one to de-allocate whole vectors from the Bloom filter if need be. Asymptotically it has the same performance as the traditional Bloom filter provided the shorter vectors are still large (Kirsch & Mitzenmacher, 2008).

2.3.5 The WISARD architecture

Igor Aleksander’s WISARD architecture is also relevant to our work (Aleksander *et al.*, 1984). In WISARD, a small number of randomly-chosen bits in x are grouped together to form an index function which indexes z . To initialise the network we create many such index functions and set z to the zero vector. When we store a pattern we calculate the indexes into z and set those entries to 1. As described the WISARD architecture sounds very similar to the Bloom filter (both ours and the traditional form), however the WISARD architecture is designed to recognise similar patterns. This forces the hash functions to be very simple and a typical hash function would take the form:

$$h_k(x) = 16x_{22} + 8x_{43} + 4x_6 + 2x_{65} + x_{75}. \quad (2.16)$$

This partitions z into blocks of 32 bits and we would have a different partition for each function. If we want to test whether a similar pattern has already been stored we again calculate the hash functions for the pattern but now we count how many bits have been set at the specified locations. If the pattern is similar to one already stored then the number of bits set will be correspondingly higher. We could emulate the WISARD functionality by turning equation 2.16 into the following set of sigma-pi functions:

$$\begin{aligned}
h_0(x) &= (\neg x_{22})(\neg x_{43})(\neg x_6)(\neg x_{65})(\neg x_{75}) \\
h_1(x) &= (\neg x_{22})(\neg x_{43})(\neg x_6)(\neg x_{65})(x_{75}) \\
h_2(x) &= (\neg x_{22})(\neg x_{43})(\neg x_6)(x_{65})(\neg x_{75}) \\
&\dots \\
h_{31}(x) &= (x_{22})(x_{43})(x_6)(x_{65})(x_{75})
\end{aligned} \tag{2.17}$$

However we have a different aim, in that we want to be sensitive to different patterns, even if they are similar. In section 2.4.1 we analyse our Bloom filter’s sensitivity to a single changed bit from a known pattern. This motivates the different form of our hash functions from the WISARD architecture.

2.4 Results

In figure 2.7 we compare the theoretical performance with the empirical performance of both the standard Bloom filter and the neurally-inspired Bloom filter. The grey shaded lines represent the theoretical performance of each and the empirical performance of both are very close to their theoretical performance. There are some regimes where the Bloom filter departs from the expected performance as a result of being forced to use a discrete number of hash functions.

To obtain these plots we stored $R = 100$ random patterns each consisting of $N = 32$ bits. We let the size of the storage range from $M = 100$ to $M = 2500$. Note that even with the maximum storage we tested both methods still used fewer bits than naively storing the patterns would use ($NR = 3200$). The h_m functions were constructed by “AND”ing $a = 10$ distinct variables together and “OR”ing $b = 10$ such terms together.

As an aside, we would need to take more care with our boolean hash functions h_m if we knew we were storing items with binary representations that had known

2. THE NEURALLY-INSPIRED BLOOM FILTER

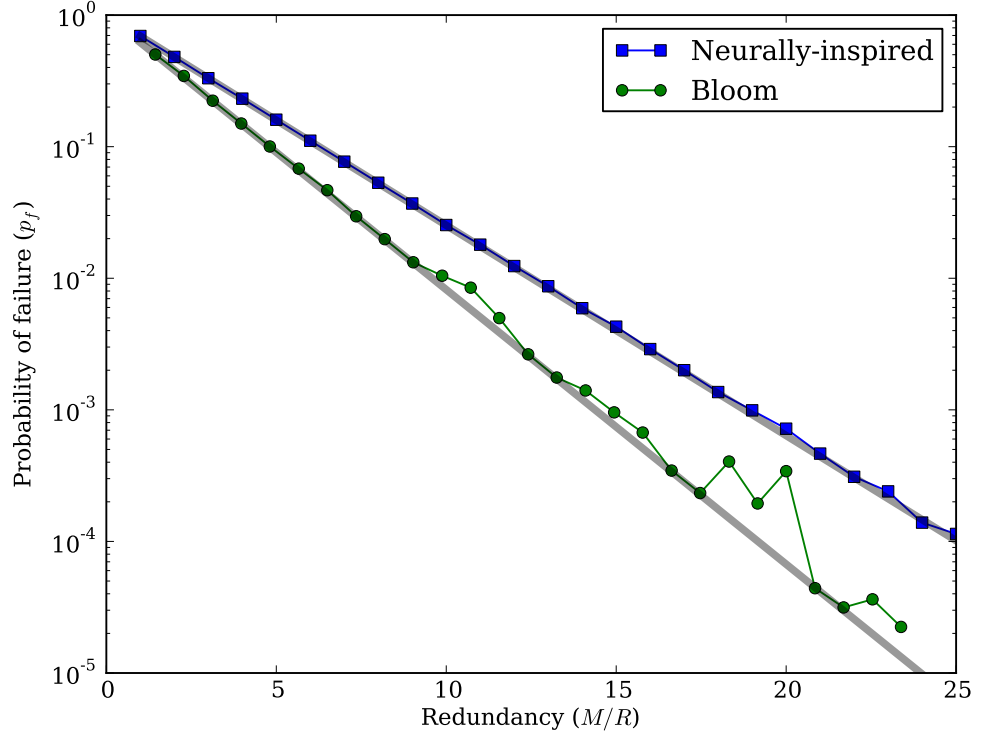


Figure 2.7: A comparison of the Bloom filter and its neurally-inspired equivalent. The grey lines give the expected theoretical performance, which closely match the experiments. The neurally-inspired Bloom filter requires approximately 31% more storage as a result of the simple bitwise-independent calculations. These figures were generated by storing $R = 100$ patterns of $N = 32$ bits each, the size of the storage varied from $M = 100$ to $M = 2500$ which is still better than naively storing the patterns ($NR = 3200$) and the improvement would increase with larger N .

correlations between pairs of bits in x , or some bits in x were biased to a particular value. The boolean functions h_m would need to be changed, otherwise some bits in the storage would be significantly more likely to be on than others and the information contained would be reduced. In the case of asymmetric bit probabilities we would need to ensure that there were the same number of negations in each “AND”ed term.

2.4.1 Sensitivity to a single bit flip

Since a single boolean hash function uses only a small number of the bits in x we might expect that for an unstored x that is very similar to a stored pattern will have a much higher probability of error than a completely random binary vector. For concreteness we will estimate the probability of error for an unstored binary vector that differs in only a single bit from a stored pattern.

For a conjunction of a terms in h_m the probability that it contains the flipped bit is $\frac{a}{N}$. If all we know is that the hash function returned false then flipping a bit that is used by the hash function will result in the hash function returning true with probability p (approximately). Therefore the probability of a previously-false hash function returning true given a single bit flip is approximately $\frac{ap}{N}$. After R patterns have been stored there will be approximately $M(1 - p)^R$ zeros in Z . The network fails to detect a bit flip when none of the zeros in z is turned on:

$$\begin{aligned} \Pr(\text{bit failure}) &= \left[1 - \frac{ap}{N}\right]^{M(1-p)^R} \\ &\approx e^{-Mp(1-p)^R \cdot \frac{a}{N}} \\ &= [\Pr(\text{failure})]^{\frac{a}{N}}. \end{aligned} \tag{2.18}$$

This approximation is only accurate when the total number of possible boolean hash functions is much larger than M the number actually used. As a thought experiment consider the extreme case when $a = 1$, $b = 1$ and the hash functions are simple bit copies (with random negation). If after storing two patterns with a different value in the n^{th} bit, then all hash functions which copy that bit will have been turned on and we will be unable to distinguish any bit flips in that position, regardless of the size of z .

Equation 2.18 captures the intuitive fact that, as more bits are “AND”ed together the network’s performance for neighbouring binary vectors becomes more independent, and in the extreme case of all the bits being “AND”ed together ($a = N$) the

2. THE NEURALLY-INSPIRED BLOOM FILTER

performance is completely independent. Practically though only a small number of bits can be “AND”ed together, as the number of “OR”s required to maintain a constant probability of returning true increases exponentially. Our results are shown in figure 2.8, and we can see that the approximations are relatively accurate. This plot was made setting $M = 2000$, $N = 32$, storing $R = 100$ patterns and we let a range from 5 to 10, while keeping the probability of returning true as close to optimal as possible. It is clear from the figure that increasing a decreases the probability of a false positive from a single bit flip.

2.4.2 Conclusion

A neurally-inspired Bloom filter, while having asymptotically the same performance, underperforms a standard Bloom filter. Moreover the neurally-inspired Bloom filter has a higher probability of error for items that are similar to already-stored items, while the standard Bloom filter has no such problem. While such negative results might cause one to discard the neurally-inspired Bloom filter, we will show that there is still an area where the neurally-inspired Bloom filter outperforms. The neurally-inspired Bloom filter has simple boolean hash functions, which allow us to perform inference cheaply over all possible x vectors. This allows us to create an associative memory with very high capacity, which we will show in the next chapter.

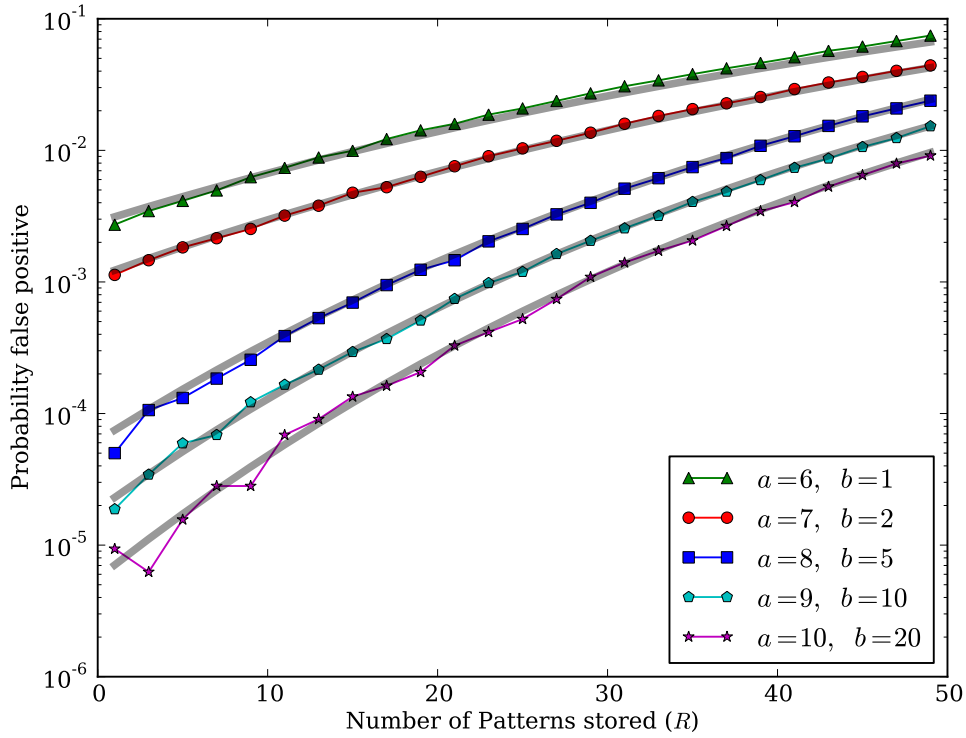


Figure 2.8: The probability of a false positive for a binary vector which differs in only one bit from an already-stored pattern. In this case increasing the number of “AND”s a decreases the false positive probability, but comes at the computational cost of exponentially increasing the number of “OR”s b . This plot was made setting $M = 2000$, $N = 32$, storing $R = 100$ patterns. If we were to increase N the probability of error would increase. The grey lines are error rates predicted by equation 2.18.

2. THE NEURALLY-INSPIRED BLOOM FILTER

CHAPTER 3

Binary associative memory

In this chapter we turn the neurally-inspired Bloom filter into an associative memory. An associative memory is similar to a Bloom filter in that both store many patterns, but the associative memory is required to return much more information. For a Bloom filter we provide a pattern and the Bloom filter answers whether or not it has been seen previously. For an associative memory, an approximate pattern is provided and the associative memory returns the closest stored item.

As an example of the associative memory task consider being shown photos of different people's faces and given some time to memorise them. Later you are presented with a blurry, partial photo of a face and you must identify the face and be able to sketch in any missing or indistinct features in the photograph. More formally, if one stores several patterns in an associative memory one can then query the memory with a corrupted version of the pattern, or cue. If the associative memory is functioning correctly it will be able to return the corresponding uncorrupted pattern.

We present two variations of the associative network. In the first there is an ideal number of patterns to store and if extra patterns are stored the total information that one can recall will gradually decrease back to zero. In the second network there is no such limit, as more patterns are stored the older patterns are gradually forgotten. We name the first network the neurally-inspired associative memory, and the second the neurally-inspired palimpsest network.¹

This chapter also presents a method to measure the performance of binary associative memory. Provided the memory makes symmetric errors, i.e. it is as likely to turn a 1 into 0 as a 0 into 1, then we can use results from the binary symmetric channel

¹A palimpsest refers to a medieval manuscript where the original writing has been washed or scraped off and a new text written in its place. Occasionally this happened to a manuscript repeatedly as parchment was both durable and expensive.

3. BINARY ASSOCIATIVE MEMORY

(MacKay, 2003) to quantify the amount of information in the recall. To measure the information that is added by the memory we subtract the information that was already present in the initial cue. This leads naturally to a performance trade-off between the reliability of the cue and the amount of information filled in. Very reliable cues provide strong evidence for the inference of the rest of the cue, but leave little room for improvement of the cue. We find that there is an optimal level of noise for the cue (but this varies between different networks and parameter settings).

The associative memories presented here have the attractive property that we are able to remove computational units from the network at any time and the network will still function as before, but with a gradual increase in the probability of bit errors. The palimpsest network can also have computational units added to it at any time. When units are newly added they may cause a slight increase in the probability of bit error in the short-term, but will improve the performance in the long-term.

3.1 Neurally-inspired associative memory

The neurally-inspired associative memory is constructed directly from the neurally-inspired Bloom filter of the preceding section. Naïvely one could iterate through all possible binary vectors and use the Bloom filter to rule out most of them. We could use the noisy cue as a prior to assign preferences to the remaining vectors. To predict a single bit in x the ideal Bayesian approach is then to marginalise out over all possible settings of the other bits in the binary vector. Such an approach is computationally infeasible, but we now present a message-passing algorithm which can use the simple, local boolean hash functions and is able to approximate the ideal algorithm. This message-passing algorithm is based on the sum-product algorithm which is now described in detail and forms an integral part of both associative memories.

3.1.1 The sum product algorithm

The sum-product algorithm is a means of efficiently calculating the sum of many products. This occurs frequently in Bayesian inference as one needs to marginalise out many irrelevant (but necessary) variables. The dependencies in the Bayesian model can be identified through the use of a factor graph. Identifying which variables are

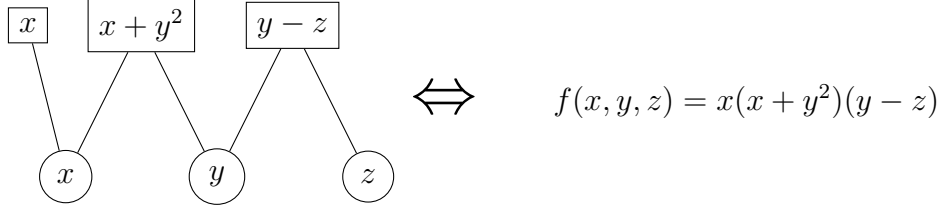


Figure 3.1: An example factor graph for the function $f(x, y, z) = x(x + y^2)(y - z)$.

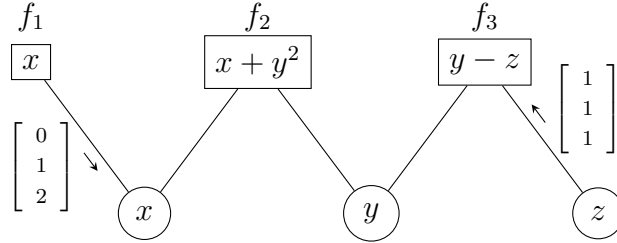
conditionally independent allows one to perform the marginalization much more efficiently. Since multiplication distributes over addition it is possible to sum over individual factors instead of over the entire function which can be exponentially more expensive.

A factor graph represents a function which is composed of many factors. It is a bipartite graph consisting of all the variables and factors which make up the function. If a particular variable appears as part of a factor then the graph has an edge connecting the variable and the factor. To highlight the bipartite nature we represent variables with circular nodes and factors with rectangular nodes. While edges only ever occur between variables and nodes, if a single variable only occurs within two factors then we will sometimes simplify the graph and only show a connection between two factors (as is done in figure 3.3(b)). It is important to remember that there is an implicit variable which is not being explicitly shown.

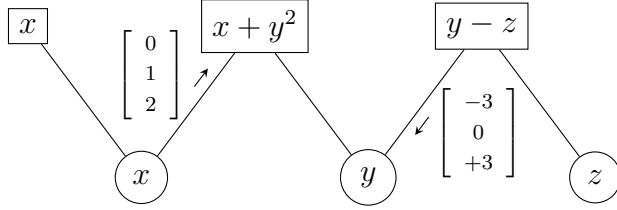
Factor graphs are useful when one wishes to sum the function over many different settings of the variables. In figure 3.1 there is an example factor graph for the function $f(x, y, z) = x(x + y^2)(y - z)$. As an example exercise we will assume that x , y and z can take values in the set $\{0, 1, 2\}$ and calculate $\sum_{xyz} f(x, y, z)$ using the sum-product algorithm as shown in figure 3.2.

In the sum-product algorithm there are two types of messages which are passed in the factor graph, those going from variables to functions and those going from functions to variables. When calculating a message from a variable to a function, we take the product of all other incoming messages:

3. BINARY ASSOCIATIVE MEMORY

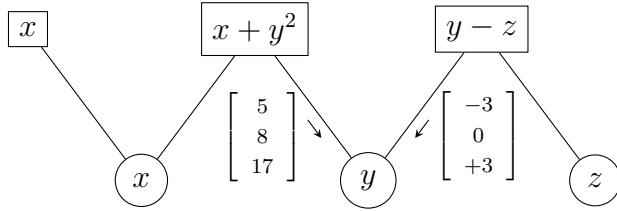


(a) Functions of only one variable transmit their values directly: $m_{f_1 \rightarrow x}(x) = f_1(x)$. Variables that occur in only one function transmit the all-ones message: $m_{z \rightarrow f_3}(z) = 1$.



(b) As a result of equation 3.1 sending a message from a function to a variable (x in this example) which occurs in only two factors will simply send the message to the other factor. The other message is calculated using equation 3.2:

$$m_{f_3 \rightarrow y}(y) = \sum_z f_3(y, z) m_{z \rightarrow f_3}(z).$$

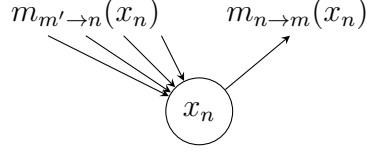


(c) To calculate the final answer at variable y we multiply all the incoming messages together and sum:

$$\sum_{xyz} f(x, y, z) = 5(-3) + 8(0) + 17(3) = 36.$$

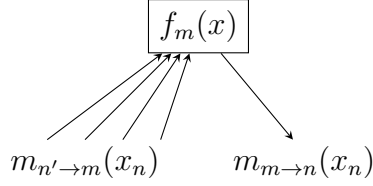
Figure 3.2: Using the sum-product algorithm to calculate $\sum_{xyz} f(x, y, z)$, assuming x , y and z only take values in the set $\{0, 1, 2\}$. Since the variables are constrained to take only three values any message from or to a variable can be represented by three numbers.

3.1 Neurally-inspired associative memory



$$m_{n \to m}(x_n) = \prod_{m' \neq n} m_{m' \to n}(x_n) \quad (3.1)$$

The more complex messages are the ones from functions to variables. One needs to sum the function over all possible binary vectors multiplied by the messages from the other variables. The number of terms to sum over for a general function would be exponential in the number of variables of that function. However there are many useful functions for which the summation can be simplified and this dissertation will only examine those functions.



$$m_{m \to n}(x_n) = \sum_{x_{\setminus n}} \left(f(x) \prod_{n' \neq n} m_{n' \to m}(x_{n'}) \right). \quad (3.2)$$

When using the sum-product algorithm to perform inference in binary variables there are generally some simplifications. Since the probabilities must sum to one we can renormalise any messages from variables (although this leaves us unable to calculate the evidence for the model, it generally leads to more numerically robust algorithms). One can also convert from binary to the log-odds representation:

$$l_i = \log \frac{p_i}{1 - p_i} \quad (3.3)$$

and then the processing in equation 3.1 is simplified to normal addition. This is used in later chapters for easier analysis of some of the algorithms.

For a more detailed explanation of the sum-product algorithm see Kschischang *et al.* (2001) or Hennig (2011). Unfortunately the sum-product algorithm is only exact for graphs which form a tree, and all the applications considered here are not trees as they have loops in the graph. This requires the use of loopy belief propagation.

3. BINARY ASSOCIATIVE MEMORY

3.1.2 Loopy Belief Propagation

The sum-product algorithm can be extended to perform inference in loopy graphs. It is then known as loopy-belief propagation. Messages have the same form as in the sum-product algorithm. However deadlock will occur if a message is only passed from a node when all other messages have been received¹. To avoid the deadlock all messages can be initialised to unity (which does not affect the product). In this manner there is no deadlock and all messages can be passed. However the messages will initially not be correct and will need several updates before all relevant information has spread through the graph. In practice the updated messages need to be averaged with the preceding message to avoid instability. Even with averaging there is no guarantee of convergence, though empirically the results have been well-behaved. Throughout this dissertation loopy belief propagation is run until convergence or a fixed number of iterations have been exceeded.

Loopy belief propagation (Pearl, 1988) can be viewed as an approximation to the real posterior (MacKay, 2003) and empirically gives good performance in many cases (Murphy *et al.*, 1999). The approximation appears to get worse as the loops get shorter and more numerous (Weiss, 2000). Fortunately for our applications the average loop length will increase as one stores longer patterns (N the number of bits in x). Performing inference on the conjunction of many independent variables ensures that only a small number of factors contribute meaningfully to the inference so, even though the graphs considered in this chapter are very loopy, the inference appears empirically accurate.

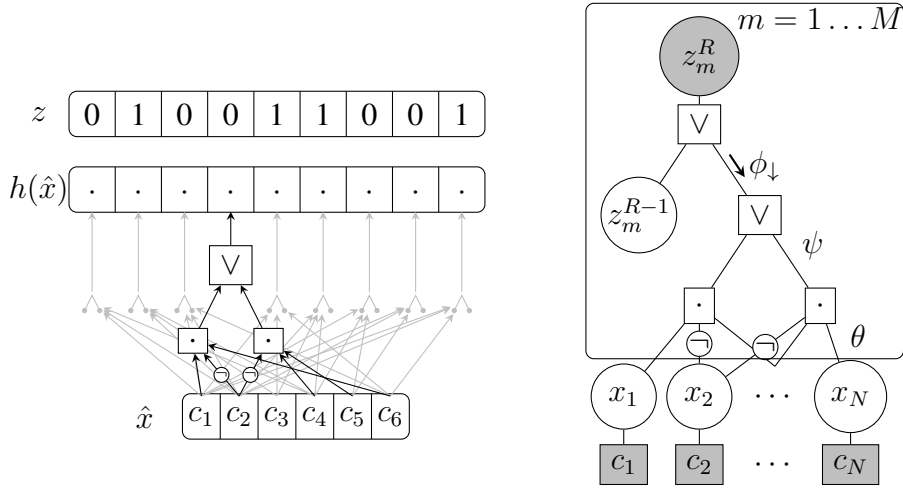
3.1.3 Recall

Once all the patterns are stored in the network we would like to retrieve one, given an input cue in the form of the marginal probabilities of the bits in x being on: $\Pr(\hat{x}_n = 1) = c_n$.

If we cared about recalling the entire x correctly and viewed a single bit error as a failure of the network, then we would want to find the most likely x vector. However in this dissertation we tolerate a small number of bit errors in x . To minimize the

¹A loop in the factor graph means that a node will be waiting for a message which in turn depends on the node itself.

3.1 Neurally-inspired associative memory



(a) After many patterns have been stored in the neurally-inspired Bloom filter we would like to recall a stored pattern that is similar to a given cue c , we calculate the bitwise marginal posteriors $\Pr(x_n | c, z)$.

(b) The factor graph for inference during recall. We use ϕ , ψ , and θ as shorthand for the various stages in the message algorithm. The ϕ 's are calculated once while the updates for ψ 's and θ 's are alternately calculated until convergence or a maximum number of iterations has been reached.

Figure 3.3: The associative memory task and the factor graph that achieves that task.

3. BINARY ASSOCIATIVE MEMORY

probability of bit error for bit x_n we should predict the most likely bit by marginalising out everything else that we believe about the rest of x . Our posterior over all possible binary strings for x after observing the noisy z is given by Bayes' theorem (Sommer & Dayan, 1998):

$$\Pr(x | z) = \frac{\Pr(x) \cdot \Pr(z | x)}{\Pr(z)}, \quad (3.4)$$

where $\Pr(x)$ represents the prior probability of x . We also assume for simplicity that any noise in x is independent and all bits are flipped with identical probability (more complicated situations can be handled by the method presented here, but this adds nothing to the discussion). To make a prediction for an individual bit x_n we need to marginalise out over all other bits $\Pr(x_n | z) = \sum_{x_{\setminus n}} \Pr(x | z)$ (where $x_{\setminus n}$ refers to changing all possible values of x except the n^{th} value).

$$P(x_n | z) \propto \Pr(x_n) \left(\sum_{x_{\setminus n}} \prod_{i \neq n} \Pr(x_i) \prod_m \Pr(z_m | x) \right) \quad (3.5)$$

The marginal calculation above can be carried out using the modified sum-product algorithm known as loopy belief propagation. In general we take the problem (as shown in figure 3.3(a)) and turn it into a factor graph (as shown in figure 3.3(b)). The solution can then be found using message passing along the edges of the factor graph. Two messages need to be passed for each edge - one message going up and one down.

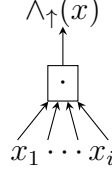
3.1.4 Calculation of Messages

We will use the index m to refer to functions since we will typically have M encoding functions, and use the index n to refer to the variables (although there may be more than N variables, if we wish to calculate other quantities of interest). All of the messages communicated here are for binary variables and consist of two parts. As a shorthand we denote the message $m(x_n = 0)$ by \bar{x}_n and $m(x_n = 1)$ by x_n . The simplest function to consider is negation:

$$\neg \begin{bmatrix} \bar{x}_n \\ x_n \end{bmatrix} = \begin{bmatrix} x_n \\ \bar{x}_n \end{bmatrix}. \quad (3.6)$$

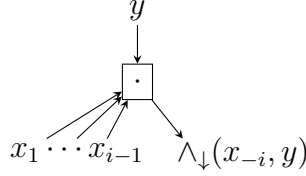
Effectively a negation interchanges the two messages, and is symmetric in either direction. If we want to consider the “AND” of many variables $\{x_n\}$, the up message is:

3.1 Neurally-inspired associative memory



$$\wedge_{\uparrow}(x) = \left[\frac{1 - \prod_i x_i}{\prod_i x_i} \right]. \quad (3.7)$$

However the “AND” message is not symmetric in either direction and has a different form for the down message:



$$\wedge_{\downarrow}(x_{-i}, y) = \left[\frac{\bar{y}}{\bar{y} + (y - \bar{y}) \left(\prod_{j \neq i} x_j \right)} \right] \quad (3.8)$$

The “OR” message can be derived from the observation that:

$$x_1 \vee x_2 = y \quad (3.9)$$

is logically equivalent to:

$$(\neg x_1) \wedge (\neg x_2) = \neg y, \quad (3.10)$$

which gives:

$$\vee_{\uparrow}(x) = \left[\frac{\prod_i \bar{x}_i}{1 - \prod_i \bar{x}_i} \right] \quad (3.11)$$

and

$$\vee_{\downarrow}(x_{-i}, y) = \left[\frac{y + (\bar{y} - y) \left(\prod_{j \neq i} \bar{x}_j \right)}{y} \right]. \quad (3.12)$$

It is important to note that all these messages are calculated in a time that is linear in the number of variables. Linear complexity is far better than might be naively expected from a method that requires calculating the sum of exponentially many terms. Fortunately the structure in “AND” and “OR” provides computational shortcuts.

In figure 3.3(b) the factor graph for the associative memory is shown. The down message for ϕ is:

$$\begin{bmatrix} \bar{\phi}_m \\ \phi_m \end{bmatrix} = \begin{bmatrix} 1 - z_m p^{R-1} \\ z_m \end{bmatrix}, \quad (3.13)$$

3. BINARY ASSOCIATIVE MEMORY

and only needs to be calculated once. We initialise $\theta_m = 1$ and $\psi_m = 1$, and then alternate calculating the θ 's then the ψ 's until the updates have converged. Since this is loopy belief propagation there is no convergence guarantee so we also quit if a maximum number of iterations have been exceeded. The final marginals are calculated by:

$$\Pr(x_n | z, c) \propto c_n \cdot \prod_m \theta_{m \downarrow n} \quad (3.14)$$

Here we return a binary vector as this allows us to compare our method with other non-probabilistic methods. We do this by maximum likelihood thresholding:

$$\hat{x}_n = \mathbb{1} [\Pr(x_n | z, c) > 0.5]. \quad (3.15)$$

We now estimate the probability that this procedure results in a bit error. If we have a sufficiently accurate prediction of the probability of error then we can use those predictions to find the parameters of the network that give optimal performance.

3.1.5 Theoretical estimate of the bit-error

Having a theoretical model of the probability of bit-error for neurally-inspired associative memory allows us to set the network parameters so as to allow maximum recall. See figure 3.4 for a diagram of the three main error conditions when recalling the target pattern and these are:

1. there could be false positives from the Bloom filter,
2. another pattern which is similar to the target pattern could have been stored,
3. there could be single-bit errors in the Bloom filter for the target pattern.

We can estimate the final probability of error as the union of these three types of errors (and assuming the errors are independent of each other). Firstly for the Bloom filter we know the probability of a false positive for a random binary vector (given in equation 2.6). The uncertainty in the cue means that we expect the answer to have approximately Np_c bits flipped and there are approximately $\binom{N}{p_c N}$ such binary vectors. The probability of no false positives in the entire set of possible vectors is:

$$\xi_1 = \Pr(\text{No false positive}) = \left(1 - e^{-Mp(1-p)^R}\right)^{2^{NH_2(p_c)}}, \quad (3.16)$$

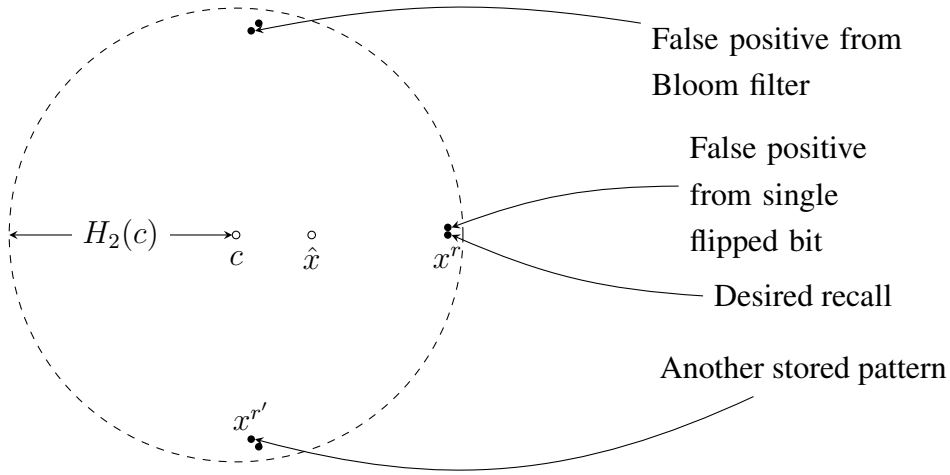


Figure 3.4: The uncertainty in the input cue determines a ball of possible binary vectors to recall. When N is large almost all of the mass of this high-dimensional ball will be at the edges. The Bloom filter is able to rule out almost all of the binary vectors. In this diagram we show all possibilities that could remain: the desired pattern x^r , another pattern $x^{r'}$, and any false positives the Bloom filter is unable to rule out. Loosely speaking, the returned estimate \hat{x} will ideally be the average of the correct recall with any false positives. Since the Bloom filter operates in a bit-wise fashion, if a binary vector is an element of the Bloom filter, then its neighbours are more likely to be false positives, so we should expect to find clumps of false positives around stored patterns.

3. BINARY ASSOCIATIVE MEMORY

where we have used the approximation $NH_2(p_c) \approx \log_2 \binom{N}{p_c N}$. Secondly we also calculate the probability that no other pattern falls within the expected radius of the cue (Kanerva, 1988):

$$\xi_2 = \Pr(\text{No other stored patterns}) = (1 - (R - 1)2^{-N})^{2^{NH_2(p_c)}}. \quad (3.17)$$

The final error condition occurs not in the large ball of potential binary vectors, but instead in the small region around the desired pattern to be recalled. A false positive is much more likely for binary vectors that differ only by a single bit from a known pattern. This probability of error is derived in equation 2.18 (on page 27):

$$\xi_3 = \Pr(\text{bit failure}) \approx \left(e^{-Mp(1-p)^R} \right)^{\frac{a}{N}} \quad (3.18)$$

While this probability of error can be decreased by increasing a (the number of “AND”s) it also requires exponentially increasing b (the number of “OR”s) and very rapidly becomes impractical. However if the initial cue differs from the stored pattern in the bit that is also accepted as a false positive, then it is very likely that the bit will be incorrect in the final recall.

We estimate the information transmitted by assuming that if any of the three failure conditions occur, then the probability of bit error reverts back to the prior probability:

$$\Pr(\text{final bit error}) = \Pr(p_c) \cdot \Pr(\xi_1 \vee \xi_2 \vee \xi_3). \quad (3.19)$$

This is a drastic simplification as it assumes there are only three possible sources of errors. There are other sources, for example there is no guarantee that performing inference on the marginals of each x_n is as informative as considering the exponentially many x vectors in their entirety. Moreover the inference is only approximate as the factor graph is loopy.

The most relevant literature for this section is Kirsch & Mitzenmacher (2006), who consider the task of determining whether an item is similar to any item already stored, or significantly different from all other items. They do this by creating a partitioned Bloom filter with a distance-sensitive hash function. If a similar item has already been stored, then there will be a significant number of ones in the locations specified by the hash functions. Their approach requires a large amount of storage and they can only tell the difference between items very near or very far (in the paper “near” is described as

an average bit-flip probability < 0.1 and “far” is an average bit-flip probability > 0.4). We note in passing that our method should also be able to provide a rough estimate of the distance from the nearest stored item. This can be achieved by not normalising the variables at each step and using the final marginals to estimate the total probability mass in the high-dimensional ball of typical vectors, although we do not explore this any further.

We will compare the theoretical predictions to actual experiments shortly. We must first consider how to measure the recall performance given the noisy cue and noisy recall and we do this in the next section.

3.1.6 Measuring Recall Performance

Assume a noisy cue is fed to an associative memory, which returns a (potentially noisy) recall; we would like to evaluate how much information the associative memory has provided. The most relevant capacity measures are derived from results in information theory (Knoblauch *et al.*, 2010). The random negations of terms in the conjunctions mean that the errors will be symmetric (i.e. if there is an error it is equally likely that a one has flipped to a zero as a zero flipped to a one). The binary symmetric channel (Richardson & Urbanke, 2008) is then close to an ideal model of the information contained in the recall (note that in reality there are dependencies between errors in x_n which violate the binary symmetric channel assumptions, although in practice they appear to have negligible effect).

The amount of information in a binary vector with symmetric noise p is defined to be (MacKay, 2003):

$$s(p) = N \cdot (1 - H_2(p)), \quad (3.20)$$

where H_2 is the standard binary entropy function. We use s to help remind us that we are measuring signal rather than noise. Since we are using a fully probabilistic method to infer the bits, we might be tempted to use those posteriors to estimate the signal. This would be incorrect because loopy belief propagation only calculates approximate posteriors; instead we use the empirical errors to estimate the information.

It is also important to take into account the fact that there is information in the cue. Generally other networks that suffer from catastrophic failure simply define the capacity as the average number of patterns that can be stored before such failure occurs.

3. BINARY ASSOCIATIVE MEMORY

Other networks look at the number of patterns that can be stored before the average error rate is worse than a predefined threshold. However this threshold is arbitrary and the performance also depends on the amount of noise present in the cue. Much associative memory work avoids this dependence by using noiseless cues. However we take the philosophical view that associative memory is only useful when it is able to *improve* an initial *noisy* cue. We measure the information provided by the network as the increase of information from the cue to the final recall:

$$\begin{aligned}
 I(p_c) &= (\text{Recalled information}) - (\text{Cue information}) \\
 &= s(p_x) - s(p_c) \\
 &= N \cdot (H_2(p_c) - H_2(p_x))
 \end{aligned} \tag{3.21}$$

This expression comes from the simplifying assumption that the reliability in each bit of the initial cue is the same (p_c). If, as an example, the associative memory were used for pattern completion where half the cue was known with certainty, and the other half completely unknown, then the information added would be $N_1(1 - H_2(p_x))$, where N_1 represents the number of unknown bits and p_x is the probability of error in the final recall of the unknown bits.

Another measure one could consider for a fully probabilistic network would be the predicted probability of the correct answer (most conveniently measured as an information content in log-probs). However this report will not consider this measure as the performance cannot be compared with non-probabilistic methods.

To show our results, plotting the probability of bit error on a logarithmic scale is not the correct way to show the performance for associative networks. A logarithmic scale implies that we care as much about reducing the bit error from 10^{-6} to 10^{-8} as reducing the bit error from 10^{-1} to 10^{-3} . However the increase in information is negligible when the error rate improves from 10^{-6} to 10^{-8} ($\approx 2 \times 10^{-5}$ bits), but significant when the bit error is reduced from 10^{-1} to 10^{-3} (≈ 0.46 bits). Where appropriate we will plot the bit-error directly in terms of information (and provide the corresponding bit error as well on the scale).

This associative memory is not affected by the order of storage. If x^1 is stored, followed by x^2 then z would be exactly the same as if x^2 were stored, followed by x^1 (as opposed to the next network where more recent patterns overwrite older patterns). If we let R be the number of patterns that have been stored in the network, on average the recall performance for all the patterns will be identical. This means that the total

3.1 Neurally-inspired associative memory

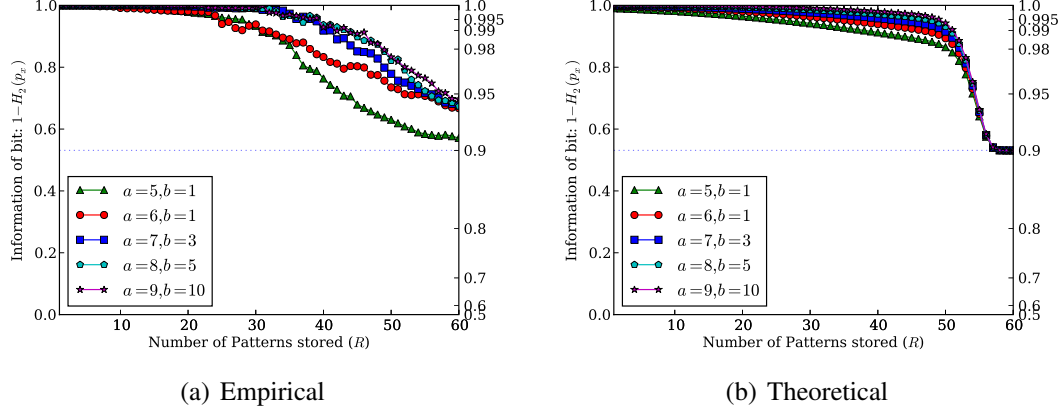


Figure 3.5: The recalled information as more patterns are stored, plotted for different numbers of terms “AND”ed and “OR”ed together in the boolean hash functions. The dotted line represents simply returning the input cue ($p_c = 0.1$), all capacities are measured as the improvement above this line. The theoretical predictions transition from reliable to unreliable far quicker than the empirical results as some types of weak statistical evidence are ignored in the theoretical model. The theoretical predictions also predict greater capacity as the inference is idealized.

information recalled is the average recall performance multiplied by the number of patterns stored and we define the total recall information as the maximum:

$$I(p_c) = \max_R [I(p_c, R) \cdot R]. \quad (3.22)$$

The total information is still a function of the cue noise and has a single well-defined maximum. We call the performance at the best noise level the capacity of the associative memory and examine it in the next section.

3.1.7 Capacity of the neurally-inspired associative memory

In figures 3.5(a) and 3.5(b) we see that while the simplification in equation 3.19 leads to a poor prediction of the bit error rates, it still predicts fairly accurately the transition from reliable to unreliable behaviour. It is at this transition that the associative memory is providing the most information so the approximations lead to acceptable predictions for the optimal performance. We created this figure using $N = 100$ and $M = 4950$

3. BINARY ASSOCIATIVE MEMORY

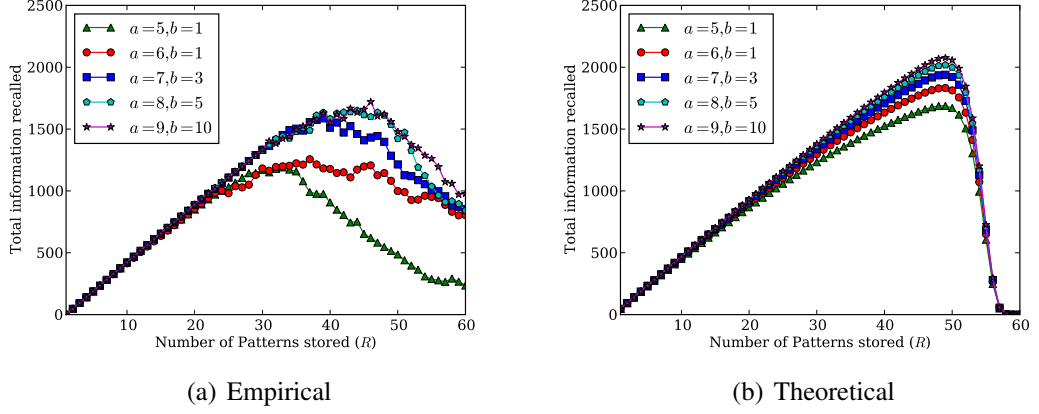


Figure 3.6: The total amount of information recallable by the network as more patterns are stored. If more patterns are stored than is optimal z becomes less informative. For this figure $p_c = 0.1$, $N = 100$ and $M = 4950$. The unusual value for M allows us to directly compare our results with Hopfield networks which are presented in a later section.

with 10% noise in the input cue. (The slightly unusual value for M allows us to compare our results directly with a Hopfield network in the next chapter.) We can also see that the performance improves as we use boolean hash functions with more variables, although the improvements rapidly become negligible (e.g. $a = 8$ and $a = 9$ are indistinguishable). Once too many patterns have been stored, the theoretical prediction is that the performance rapidly reverts to the prior performance. Empirically the performance degrades far more slowly. This is due to the assumption (made in equation 3.19) that any error conditions will revert the probability of error to the prior probability; in truth we can encounter several errors and still have weak statistical evidence for the correct answer. As an example of this, false positives are far more likely to have a small number of $h_m(x)$ return true, while a stored pattern will typically have many more $h_m(x)$ return true. This difference provides weak statistical evidence, so even if there is a pattern and a false positive equidistant from the initial cue, the recall procedure will more heavily weight the correct pattern. These considerations are entirely absent from our theoretical analysis, but do not affect the reliable regime.

This error rate can be turned directly into the total information recalled by the

3.1 Neurally-inspired associative memory

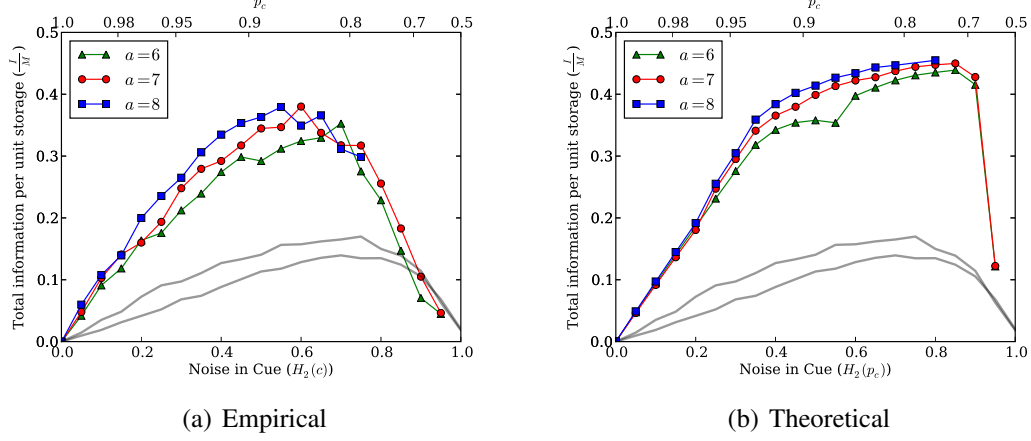


Figure 3.7: The maximum information $I(p_c)$ as a function of the cue noise. For this figure $N = 100$ and $M = 4950$. The grey lines represent the performance of a Hopfield network of comparable size, using two different decoding strategies (which are covered in the next chapter). The discontinuity in the theoretical prediction is as a result of the discrete nature of the boolean hash functions.

network. This is shown in figure 3.6. We can see that empirically the maximum information recalled occurs when $R \approx 45$ and roughly 1700 bits are recalled out of a possible 5000 (from the previous figure the final error probability $p_x \approx 0.01$). The predicted total information is greater than 2000 bits and we conjecture that the difference is due to the inference on marginals not being as informative as examining all possible binary vectors in their entirety.

In figure 3.7 we can see that our definition of the capacity depends on the reliability of the cue. The optimal parameters for the boolean hash functions also vary with cue reliability. We have plotted the hash functions with the same number of “AND”s (a) together, even though the number of “OR”s differ. At the optimal cue reliability the neurally-inspired associative memory has an efficiency of **0.36** bits per bit of storage. We refer to this as the capacity of the neurally-inspired associative memory. By comparison, in the next chapter we show the capacity of a traditional Hopfield network is approximately 0.14 bits per unit of storage, which with smart decoding can be enhanced to **0.17** bits per stored integer. This is an unfair comparison however as Hopfield networks are able to store integers, rather than a single bit; so the associative

3. BINARY ASSOCIATIVE MEMORY

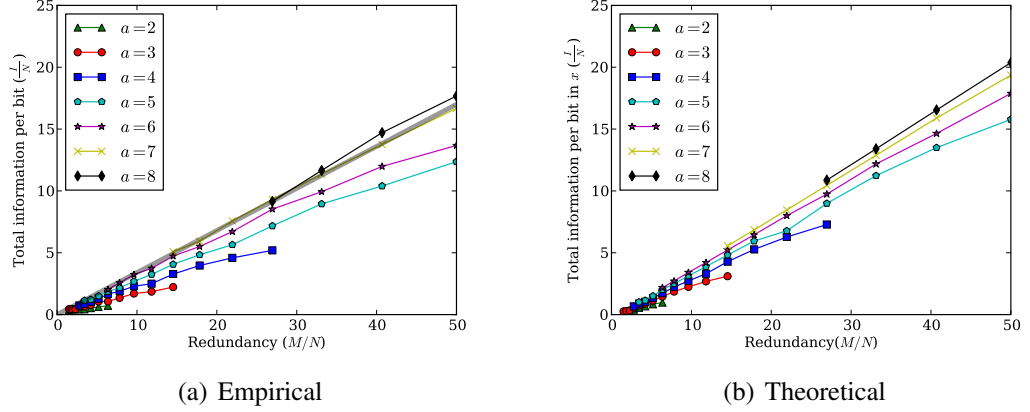


Figure 3.8: The optimal capacity of the network is linear in the redundancy. The grey line in (a) represents a linear fit using an efficiency of 0.34 bits of recall per bit of storage.

memory’s performance is even more impressive.¹

In figure 3.7(b) the theoretically optimal performance occurs for fairly noisy cues ($p_c \approx 0.3$). We believe that the inference is impaired for such noisy cues and in figure 3.7(a) the empirical performance is best for cleaner cues ($p_c \approx 0.1$).

In figure 3.8 we show that the capacity increases approximately linearly as the amount of storage increases. This scaling relationship assumes that the cue noise is kept fixed as the amount of storage is varied. For large cue noise (e.g. $p_c = 0.5$) the memory can only hope to recall a single memory and the linear scaling relationship no longer holds.

The neurally-inspired associative memory has an optimal number of patterns to store. Once this optimum is exceeded, the false positive rate of the underlying Bloom filter increases and the recall performance of the memory decreases. Another type of associative memory – the palimpsest memory – does not suffer this limitation. Instead it gradually forgets old patterns as new patterns are stored. In the next section we

¹In defense of the Hopfield network, there is work by Palm & Sommer (1992) which suggests that the performance of the clipped Hopfield network degrades only marginally. The clipped Hopfield network only uses the sign of the correlation; i.e. an element in z can only take values $\{+, 0, -\}$. However in an implementation of the clipped Hopfield network one would still need to remember the correlation so that when a new pattern is added the sign of the correlation can be correctly updated.

describe a palimpsest memory which is closely related to the memory just described.

3.2 Neurally-inspired palimpsest network

We now consider a straightforward modification to the neurally-inspired associative memory which allows gradual forgetting of old memories as new memories are added (other associative memories, most notably the Hopfield network, can suffer catastrophic failure when too many memories are stored). This type of associative memory is also known as a palimpsest network (Sandberg *et al.*, 2000). Other palimpsest networks have been proposed before (Amit & Fusi, 1994; Kanerva, 1988; Storkey, 1998; Willshaw, 1971), however our network is unique in using a noise-free storage function and principled statistical inference to achieve high capacity.

Informally the idea behind the neurally-inspired Palimpsest memory is to use the boolean hash functions from the previous section to decide where to encode a representation, while using a simple encoding function to decide what to store (see figure 3.9 for an overview of the process). We can again use distributed message passing to perform inference on what has been stored since both the encoding and decoding functions are functions of a small number of variables, and each function is evaluated independently of each other. We now describe the process in more detail.

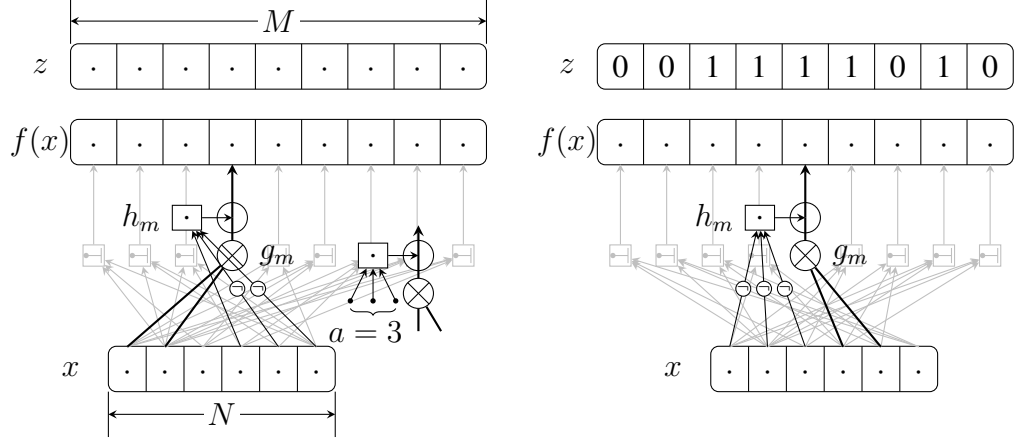
3.2.1 Storage in the palimpsest network

We use x to denote the N dimensional binary pattern being stored, with z representing the M dimensional binary storage. We initialise z to a random binary string. The computational units, one associated with each bit in z , each have a boolean hash function $h_m(x)$ and an encoding function $g_m(x)$. For the m^{th} computational unit to store a bit from x we calculate $h_m(x)$ and if the hash function returns true then we set $z_m = g_m(x)$. As a shorthand we define $f(x)$:

$$f(x) = \begin{cases} \perp & \text{if } h(x) = 0 \\ g(x) & \text{if } h(x) = 1 \end{cases}, \quad (3.23)$$

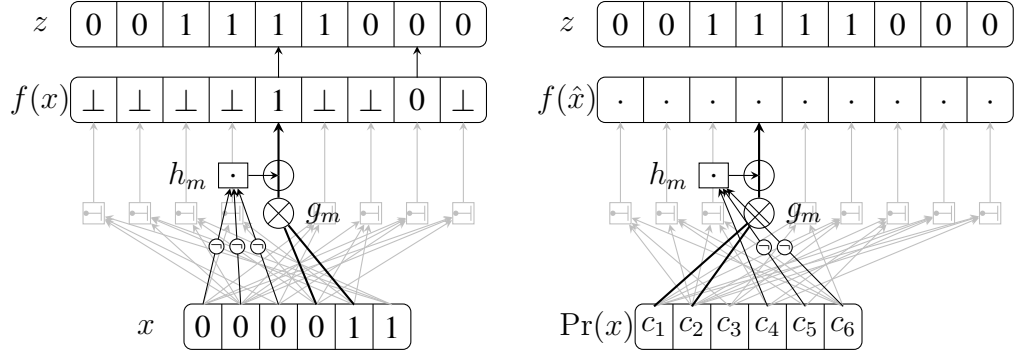
We use \perp to denote that no value is returned. There are several options for the encoding function $g(x)$; the simplest is to copy a randomly-chosen bit in x . We found slightly improved performance if the encoding function is the parity of two randomly chosen

3. BINARY ASSOCIATIVE MEMORY



(a) The structure of the neurally-inspired palimpsest memory. Each bit in z has an associated hash function h_m and an encoding function g_m of the input x .

(b) z is initialised to a random binary string.



(c) To store a pattern x , we calculate $g(x)$ and $h(x)$. We set $z_m = g_m(x)$ if and only if $h_m(x)$ returns true.

(d) Given a cue c we calculate the posterior $\Pr(x | c, z)$.

Figure 3.9: An overview of the neurally-inspired palimpsest network. If the hash function $h_m(x)$ returns true, then z_m is overwritten by the encoding function $g_m(x)$. In this section we use the same boolean hash functions from the previous section, and use the parity of two randomly chosen bits as the encoding function g_m .

bits in x . (We cover the parity function and its associated message-passing in chapter 5.)

If all we care about is ensuring that, after R patterns have been stored, the maximum number of bits from the first pattern have not been overwritten, then the probability of the hash function returning true, p_h , is uniquely defined. We start with the number of bits set when the first pattern is stored: Mp_h . The set bits will be gradually overwritten by succeeding patterns. With each pattern that is stored the proportion of original bits will decrease by a factor $1 - p_h$. After another $R - 1$ patterns have been stored there will only be:

$$Mp_h(1 - p_h)^{R-1} \quad (3.24)$$

original bits. To maximize this we set $p_h = \frac{1}{R}$.

Empirically we found that for optimal performance the number of uncorrupted bits needed to be on the order of N . This gives us a simple scaling relationship:

$$N \approx M \cdot \left(\frac{1}{R}\right) \cdot \left(1 - \frac{1}{R}\right)^{R-1}, \quad (3.25)$$

which for large $\frac{M}{N}$ can be approximated by:

$$R \approx \frac{M}{Ne} + 1. \quad (3.26)$$

There is also other work which finds a reduction in capacity by a factor of e (Mézard *et al.*, 1986; Sterratt & Willshaw, 2008; Willshaw, 1971), which suggests that there is a fundamental reduction in capacity when creating a simple palimpsest in this manner. Now the optimal probability of the hash function can be written as:

$$p_h \approx \frac{Ne}{M + Ne} \quad (3.27)$$

3.2.2 Retrieval

For the time being we assume that we know how many patterns have been stored since the pattern we are trying to recall. This means we know the probability of a bit being overwritten and hence the probability of a bit having the correct value: η^* . The

3. BINARY ASSOCIATIVE MEMORY

corresponding factor graph is shown in figure 3.10. If the noise level is fixed then the messages from observing z_m are:

$$\eta_{\downarrow} = \begin{bmatrix} z\eta^* + (1-z)\bar{\eta}^* \\ z\bar{\eta}^* + (1-z)\eta^* \end{bmatrix}. \quad (3.28)$$

We then need to do message passing to estimate the probability that h_m has returned true and also estimate the value of g_m , based on our current beliefs. The message passing for both g_m and h_m appear in sections 5.2.2 and 3.1.4 respectively, and we now focus on the messages for the conditional-copy. We use α for messages relating to the encoding function and β for messages relating to the hash function:

$$\beta_{\leftarrow} = \begin{bmatrix} 0.5 \\ \bar{\eta}_{\downarrow}\alpha_{\uparrow} + \eta_{\downarrow}\bar{\alpha}_{\uparrow} \end{bmatrix}, \quad (3.29)$$

$$\alpha_{\downarrow} = \begin{bmatrix} 0.5\bar{\beta}_{\rightarrow} + \beta_{\rightarrow}\bar{\eta}_{\downarrow} \\ 0.5\bar{\beta}_{\rightarrow} + \beta_{\rightarrow}\eta_{\downarrow} \end{bmatrix}, \quad (3.30)$$

where α_{\uparrow} and β_{\rightarrow} are messages calculated from the particular encoding and hash functions respectively.

In reality we don't know the level of noise η^* in the storage for the target pattern. The level of noise varies with the age of the pattern, as succeeding patterns gradually overwrite the storage. To infer the level of noise each computational subunit would have to maintain its own estimate of the noise, through communication with all the other subunits.

Empirically we found that the retrieval performance was similar if, instead of estimating the noise level, one used the worst noise level. Using a fixed noise level eliminates the need for communication between subunits and reduces the complexity in the retrieval process. The worst noise level occurs when the R^{th} oldest pattern is recalled:

$$\eta^* \approx \frac{1 - e^{-1}}{2}. \quad (3.31)$$

Even if the performance for the worst noise level is identical to the performance for the estimated noise level one might still want to know η^* as this gives insight into how old the pattern is and also gives a better idea of how certain the recall is (with the incorrect noise level the marginals will not be accurate). We now examine the empirical performance as it gives us further guidance on how to set the parameters of the hash functions and it allows us to find the capacity of the network.

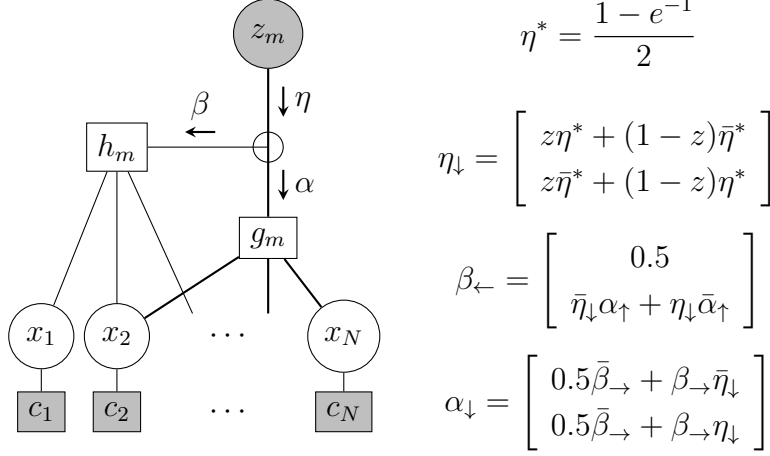


Figure 3.10: The factor graph and update equations for performing inference in the recall.

3.2.3 Capacity of the neurally-inspired palimpsest memory

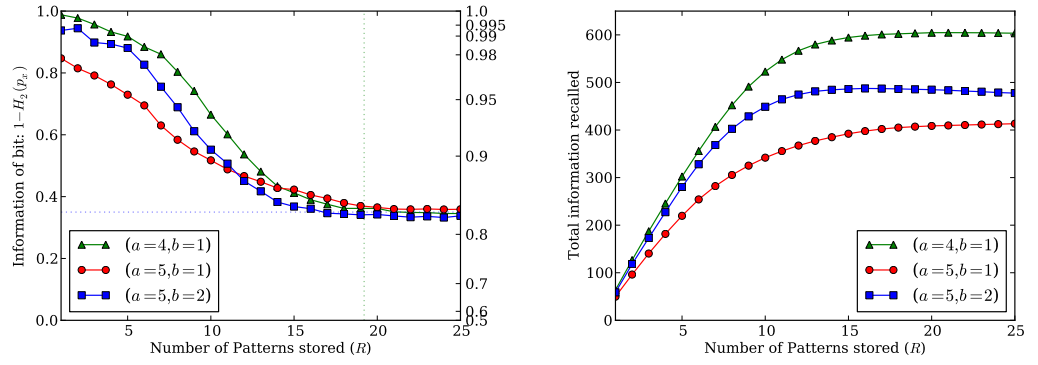
The capacity is the total information that can be recalled by the network. As recently-stored memories are recalled with better accuracy we need to sum the recalled information from all the stored patterns:

$$I(p_c) = \sum_r^R I^r(p_c), \quad (3.32)$$

where R is the total number of memories the network can recall. (This corresponds to the number of memories that can be stored before $p_x \approx p_c$.)

In figure 3.11 we show the performance for input vectors with $N = 100$ bits, a storage of $M = 4950$ bits and $p_c = 1/6$. When presented with an input cue for retrieval, roughly 17 bits are incorrect and the network needs to correct the input. The results are shown in figure 3.11. Notice that in figure 3.11(a) the predicted capacity is $R = 19$, by which time the recall performance of the original pattern has degraded to the prior performance. In figure 3.11(b) we see that hash functions of fewer variables have better performance (provided their probability of returning true is approximately p_h). The sigma-pi hash function with the fewest variables has $b = 1$ so that there is only a single conjunction of randomly-negated variables. Solving for a , the ideal

3. BINARY ASSOCIATIVE MEMORY

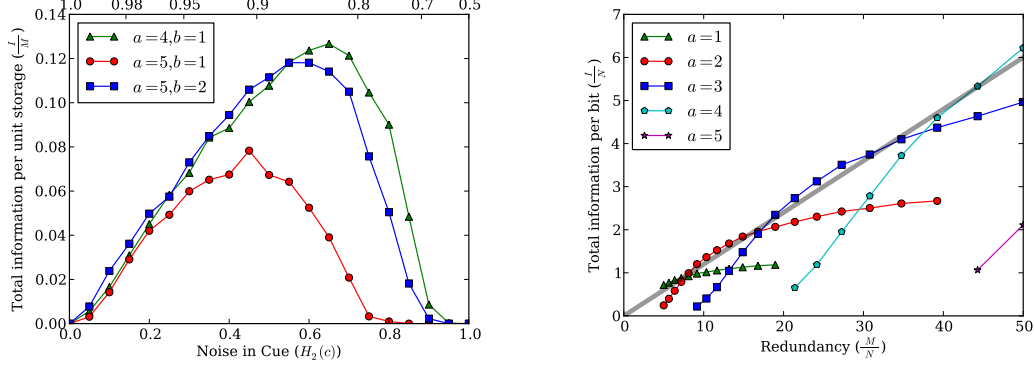


(a) The recall performance of the first pattern as more patterns are stored. As predicted, after 19 patterns have been stored the performance for all settings has reverted to the input noise.

(b) The total amount of information recalled tends to a maximum as the number of stored memories increases and older memories are forgotten. This maximum is the capacity of the network.

Figure 3.11: The performance when storing 100 bit patterns in 4950 bits of storage using a Palimpsest network. When attempting recall the initial cue contains 17% noise. Different numbers of “AND”s (a) and “OR”s (b) in h_m are plotted in different lines.

3.2 Neurally-inspired palimpsest network



(a) The capacity of the network as a function of the reliability of the initial cue (measured as the entropy of the initial cue). The best performance corresponds to (approximately) 17% noise.

(b) The capacity of the network is linear in the redundancy. The number of bits of information that can be recalled per bit in x is roughly 0.12 of the redundancy ($\frac{M}{N}$). For this figure $N = 100$ and M ranges from 500 to 5000.

Figure 3.12: The palimpsest network’s recall performance for increasing network size and increasing levels of noise in the initial cue.

number of variables to “AND” together:

$$\hat{a} \approx \log_2 \left(1 + \frac{M}{Ne} \right). \quad (3.33)$$

Figure 3.11(b) shows that the optimal $\hat{a} = 4$ is indeed the best performing configuration for the hash function $h(x)$.

In figure 3.11(b) the best network can recall approximately 600 bits using $M = 4950$ bits of storage. If we define the efficiency of our associative memory as the number of bits recalled divided by the number of bits stored then this memory has an efficiency of **0.12**. In figure 3.12(a) we show the efficiency as a function of the cue noise. The efficiency peaks for a cue with a noise level $p_c \approx 1/6$. If we fix the noise level then when we scale the storage (as shown in figure 3.12(b)) we see the amount of information that can be recalled scales linearly with the storage. The rate of increase is given by the efficiency, and we plot a grey line corresponding to an efficiency of 0.12 and find good agreement with the data.

Fusi & Abbott (2007) provides evidence that memories fade exponentially from a palimpsest network. This suggests that a network must increase in size exponentially

3. BINARY ASSOCIATIVE MEMORY

to ensure a linear increase in the lifetime of the pattern. Our results here suggests that this bad performance can be mitigated by ensuring that synapses are modified only under increasingly specific conditions.

3.2.4 Related Work

There is a large literature on associative memory. Here we briefly review the most relevant work on palimpsest networks which use binary storage.

3.2.4.1 The Palimpsest Willshaw network

Willshaw (1971) proposes a method for turning a standard Willshaw network into a palimpsest memory. As more memories are stored the saturation of z increases until performance starts to degrade. One can forget old patterns by simply resetting randomly chosen bits in z to 0. If one is careful then one can ensure that on average the optimal number of bits are on. However this is a comparatively crude technique since it relies on random resetting, while the technique presented here only overwrites old patterns with new patterns and does not introduce any extra noise. For a thorough explanation of the random-resetting technique see Henson (1993) (which also explores the probability of resetting increasing with time since the last reset, although this requires extra storage).

3.2.4.2 Stochastic storage rules

A more general approach than random resetting is described by Amit & Fusi (1994). Sparse patterns are presented to a network which has binary synapses between every pair of inputs (in this sense the network has identical connectivity to the Hopfield network which is covered in Chapter 4, but only with binary storage). The sparsity considered here is typically $\sum_n x_n = \log N$. When pattern x is to be stored in the network and both x_i and x_j are active in the sparse pattern then the synapse z_{ij} is turned on with probability q_+ . If x_i is on and x_j is off then with probability q_- the synapse z_{ij} is turned off.

The optimal transition probabilities are a function of network size (in part due to the changing sparsity of the patterns) and the number of patterns recallable in the optimal

case is proportional to N . In this sense the pattern is stored in a random location and overwrites random parts of other patterns. If a continuous stream of patterns are stored then it is shown that the distribution of on and off states converges to a fixed point. Through a careful analysis of the signal-to-noise ratio Amit & Fusi (1994) can then estimate the capacity of the memory.

Romani *et al.* (2008) use the network from Amit & Fusi (1994) but with slightly different parameters to create an efficient palimpsest familiarity memory. Barbour *et al.* (2007) use the observation that in any palimpsest memory there must be an equilibrium distribution of the storage states (or synaptic weights in biological networks). It suggests that in a palimpsest memory (which is not necessarily binary) one can learn interesting things by examining the distribution of synaptic strengths. In this chapter however all patterns are random bit vectors and the equilibrium distribution for each storage bit is a random bit.

3.2.4.3 Sparse Distributed Memory

Kanerva's Sparse Distributed Memory (Kanerva, 1988) has a lot of similarity with the work presented here. Where Kanerva refers to neurons as "address decoders", here they are described as hash functions. Kanerva also fixes the synaptic weights with random values, corresponding to our randomly chosen hash functions. Kanerva's address decoders and our binary hash functions are designed to activate for only a small proportion of input (i.e. they are sparse). Kanerva's work has an unusual level of mathematical rigour, the behaviour of the network is described with great clarity using intuitive geometric descriptions, even if the results are sometimes far from intuitive. Kanerva (1988) also makes interesting (but not compelling) comparisons between the sparse distributed memory and the cerebellum.

However the recognition functions chosen by Kanerva involve thresholding a linear sum involving the entire input pattern. Our work suggests that it is much more informative to use functions which use fewer variables. The dynamics of recognition are also just a simple linear sum for each output bit, which is not the correct Bayesian decoding. The easiest observation is that prior information (from the cue) is not retained and if too many patterns are stored then the memory is overloaded and recalls random patterns unrelated to any stored patterns. This catastrophic failure is similar

3. BINARY ASSOCIATIVE MEMORY

to the failure experienced by an overloaded Hopfield network (Hopfield, 1982) which is covered in chapter 4. A major difference is that while the Hopfield network will converge to an unrelated pattern, the sparse distributed memory will not converge but sample seemingly unrelated binary vectors. This allows one to determine whether the network is performing reliably.

A lack of proper statistical decoding means that the sparse distributed memory has a poor capacity. As an example Kanerva (1988) describes the performance of a network with $M = 10^9$ units (10^6 units for each bit in a pattern of $N = 10^3$ bits). The sensitivity is set such that 10^3 locations per bit are set for each pattern that is stored, which gives it a critical distance of 380 bits for the most recently stored pattern (in our terminology, it can clean up a cue with noise level 38%). However if 2300 more patterns are stored then the pattern has a 50% chance of converging, even if the cue is noiseless. To describe this performance using our measure would require extensive simulations. However we can get a simple upper bound on the performance by assuming that the maximally noisy cue ($p_c = 0.38$) will be successfully cleaned up with zero probability of error for all $R = 2300$ patterns. This gives:

$$\frac{RN \cdot H_2(p_c)}{M} = 2.2 \times 10^{-3}, \quad (3.34)$$

which suggests the palimpsest network presented in this chapter is at least 54 times more efficient (and the true figure is probably much higher, since the critical distance rapidly decreases with more stored patterns).

3.3 Comparison with a traditional computer

We can also compare our solution with the solution that a traditional computer would use. For the traditional Von Neumann computer architecture the associative memory task has a straightforward albeit slow solution. To store pattern r we copy the bits directly into z , taking care not to overwrite any of the previously stored memories. To recall the pattern when presented with an initial cue, one can then compare the cue to all stored patterns and return the most likely pattern. This has a high efficiency compared to the solutions in this chapter (although in chapter 5 we show that if large amounts of order and communication are allowed during recall then an associative memory can perform better than simply copying the patterns).

While this might seem a straightforward solution, we argue that there are several problems with it. The first is that this solution requires a large amount of structure; there needs to be exactly N bits set for every pattern stored, with every copied bit referring to a distinct bit in x . When recall is performed it is important to ensure that the bits are correctly partitioned into their respective sets. The comparisons with each stored pattern is naturally achieved using a central processing unit (CPU), which represents a single point of failure in the device.

The second problem with this solution is that recall has several stages. Firstly the likelihood of each pattern needs to be evaluated, then the best likelihood needs to be found and finally the winning pattern needs to be copied back into x .

The final problem with this solution is that it is brittle. If the hardware for several bits from a single pattern break then there can be a bias either for or against recalling the pattern (depending on the failure, and the exact details of the recall mechanism).

In contrast, the solutions presented here have a single stage for storage, and a single stage for recall. All processing is done independently, reducing the need for communication. There is also very little order required in the network, as the encoding functions are simply functions of randomly chosen bits of the input. The performance also degrades gracefully as hardware fails, with a gradual reversion back to the prior.

However the scaling relationship in equation 3.26 (on page 51) reveals that the palimpsest network needs to use e times as much memory as simply copying the patterns requires (and empirically we need slightly more as there is also some degradation from the approximate inference). This loss of efficiency is the trade-off for single-stage encoding and decoding with redundancy to protect against hardware failure.

3.4 Discussion

In this chapter we have presented two binary associative memories which represent patterns through the evaluation of independent, simple, random functions. These representations can then be stored in the associative memory, either by overwriting or by “OR”ing with the already-stored representations.

Using independent, simple, random functions allows us to perform inference over the whole set of stored patterns through distributed message-passing techniques, and

3. BINARY ASSOCIATIVE MEMORY

with this capability we can construct an efficient associative memory which is able to correct an initial noisy cue.

Previous associative memories generally tend to have catastrophic failure when too many patterns are stored. The point at which catastrophic failure occurs is then defined to be the capacity of the associative memory. However when sound probabilistic inference is used then catastrophic failure is avoided and the recall gracefully degrades to the original prior information, thus we need a new measure of an associative memory's capacity.

We introduced a performance measure to estimate how much information is contained in a (potentially) noisy recall. It is based on results from information theory and assumes that the bit flip probability is symmetric. Philosophically we view the *increase* of information as the important quantity (otherwise there is no reason to use an associative memory). This performance measure requires that we explore different levels of noise in the cue, but gives insight into the regime of best performance.

We have presented a high-performance associative memory which is based on the Bloom filter from the previous chapter. Empirically we found that the network would perform optimally for cues with approximately 10% of the bits flipped (which corresponds to information of approximately $\frac{N}{2}$ bits). At this optimal cue noise level the network was able to operate at an efficiency of approximately **0.36**. This means that for approximately extra 3 bits of storage the network is able to recall 1 more bit of information. Notice that even if the network had a storage of $M = NR$ bits and was able to recall R patterns perfectly then the efficiency for a cue with 10% noise would still only be:

$$\frac{NR (H_2(p_c) - H_2(p_x))}{M} \approx 0.47, \quad (3.35)$$

so the performance of the associative memory is very close to storing the patterns directly. In section 5 we will examine a more complex network where the efficiency can be pushed higher than directly storing the patterns.

We also sketched an outline of the three different types of failures that can occur in the associative memory: false positives, other stored patterns that are close to the desired recall and the Bloom filter's lessened ability to distinguish between very similar binary vectors. The message passing algorithm can be thought of as summing over all possible binary vectors that are within a radius of the initial cue. (The radius

is determined by the noisiness of the cue.) In the theoretical model there is a clear transition from reliable to unreliable recall, while in the actual network there is a more graceful decline. However the theoretical model can still guide us to areas of optimal performance.

The associative memory has a definite storage limit and if we store too many patterns the performance of the whole network will degrade. Trying to overcome this limitation led us to create a palimpsest network which gradually forgets old patterns. It does this by copying bits into locations determined by the binary vector we are trying to store. However the efficiency of the network drops to approximately **0.12** bits of recall per bit of storage.

While the efficiency of the palimpsest network might seem low in comparison to the associative network, we will show in the next chapter that it is of a comparable efficiency to the Hopfield network (in spite of the fact that the palimpsest network uses only bits for storage, while the Hopfield network uses integers). The palimpsest network is also very flexible and we can add and remove computational units (to within a factor of two) and still the network will perform near-optimally. The network is designed to tolerate noise and introducing new units is indistinguishable from adding new memories. However a large amount of inefficiency (as shown in equation 3.26) is required to obtain this flexibility. This will be required of any method which overwrites bits independently.

There are two ways around the loss of efficiency, and both require a relaxing of the independent binary calculation. Firstly we can relax the requirement that the storage contains only bits. If instead of bits we allow integers then we can create superpositions of representations by adding them together and obtain a generalization of the Hopfield network which we will cover in the next chapter. We could alternatively relax the requirement that the bits are overwritten independently. If we coordinate small populations of computational units then the small populations can be overwritten as a whole and we might expect the bulk of information gain to come from reliable recall, and we examine this in chapter 5.

3. BINARY ASSOCIATIVE MEMORY

CHAPTER 4

The Hopfield network

The previous chapter introduced a new measure with which to evaluate associative memories. We then examined the performance of two new associative memories. However it is unclear how their performance compares to other associative memories, so in this section we use our new measure to gauge the performance of one of the best-known associative memories: the Hopfield network (Hopfield, 1982), which uses the correlations of all pairs of variables to perform recall. We could have also evaluated the Willshaw network on our new performance measure. This would have the advantage that both the Willshaw network and the networks from the previous chapter use binary storage. However the Willshaw network is most efficient when storing extremely sparse patterns in contrast to the dense patterns previously used. While the performance measure could be extended to take into account the reduced information in sparse patterns, it is not clear that this would be a suitable first comparison. There might be subtle differences between storing sparse and dense patterns which invalidate the comparison. Instead we prefer to store dense patterns in a Hopfield network and compare the performance per unit of storage.

Hopfield networks store the correlations using integers. Using integers for storage will allow us to explore the efficiency of superposing representations together using addition. Regardless of the greater representational power that integers have we will show that the associative memory of the previous chapter achieves twice the capacity of a principled Hopfield network using an equal number of bits.

The traditional Hopfield network does not have a principled probabilistic basis for pattern recall, so we explore several principled variants of recall in Hopfield networks. One of the main aims of the chapter is to show that principled statistical inference gives the Hopfield network greater representational capacity and eliminates catastrophic failure. We give a simplified factor graph on which we can perform belief propagation and

4. THE HOPFIELD NETWORK

maximum a posteriori estimation and show that the update equations are almost identical to the maximum entropy method of recall. To find the capacity of the Hopfield network our performance measure requires finding the optimal level of noise and we show that Hopfield networks are most efficient (according to our metric) for cues in which approximately 20% of the bits are flipped.

We are also interested in other possible architectures that use integer storage so in the final section of this chapter we explore two other associative memories that use integers. The first considers the equality of several variables, and the second is a counting Bloom filter. Both associative memories have a neurally-inspired flavour as the connectivity is again randomly chosen and recall is performed using principled statistical inference. The encoding functions consider several variables simultaneously and as such are able to create sparse distributed representations similarly to the preceding chapter. We show that this approach of using more than two variables leads to a substantial increase in recall performance, albeit with increased decoding computational cost.

We first describe the traditional Hopfield network and provide several principled probabilistic decodings which are similar to the original recall procedure.

4.1 The traditional Hopfield network

For this chapter we use $x_n \in \{-1, +1\}$ rather than binary; this is in keeping with the standard convention of a Hopfield network and simplifies the encoding function. However instead of using the standard Hopfield network terminology of “connection weights” to refer to the correlation between a pair of inputs we will instead refer to them as storage (in keeping with previous chapters) since the connection weight contains information about the patterns that have already been stored.

The original Hopfield network (Hopfield, 1982) assumes connectivity between all possible pairs of neurons. When a pattern is presented to the network, any two neurons that have the same state have their common connection incremented, while two neurons in different states decrement their common connection.

We can split the storage procedure into two functions, an encoding function and a storage function. The encoding function takes the pattern and returns a distributed representation. The storage function takes the current storage together with a distributed

4.1 The traditional Hopfield network

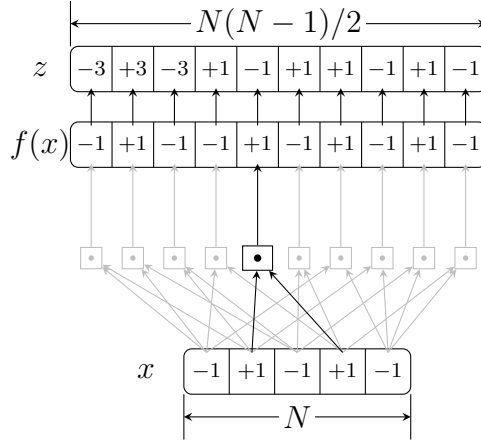


Figure 4.1: A Hopfield network can be viewed as part of our framework. Now $x_n \in \{-1, +1\}$ and $z_m \in \mathbb{Z}$. The connections also have more order as we ensure that all pairs of variables are present. The encoding is also dense as every z_m is changed when storing a single pattern.

representation and returns the values of the updated storage. As an example in the preceding chapter the neurally-inspired associative memory had a sigma-pi function as the encoding function, and used logical OR as the storage function. The encoding function for the Hopfield network is simple multiplication:

$$f_{ij}(x) = x_i \cdot x_j. \quad (4.1)$$

The storage z is now a vector of integers and we use simple addition as the storage function:

$$z_{ij}^R = z_{ij}^{R-1} + f_{ij}(x). \quad (4.2)$$

After R patterns have been stored z will have an approximately Gaussian distribution with mean $\mu = 0$ and variance $\sigma^2 = R$. In several methods we present later, this Gaussian approximation simplifies the method considerably.

The network is initialised to a starting cue (in which several of the bits are incorrect). A dynamical system is then run for several steps with each neuron updating its state according to:

$$x_i^t = \Theta \left(\sum_{j \neq i} z_{ij} x_j^{(t-1)} \right) \quad (4.3)$$

4. THE HOPFIELD NETWORK

The threshold function is defined as:

$$\Theta(y) = \begin{cases} +1; & y \geq 0 \\ -1; & y < 0 \end{cases} \quad (4.4)$$

If all neurons were allowed to update at the same time then it is possible that there would be no stable state. However if we update a single neuron at a time and cycle through all possible neurons then the system is guaranteed to converge to a stable state (through the existence of a Lyapunov function) (Hopfield, 1982). However a sequential decoding algorithm goes against the spirit of a distributed recall procedure and instead we performed the decoding in parallel. With all states updating in parallel there is no guarantee of convergence as states may oscillate after each iteration of decoding. Sequential updates may guarantee convergence, but in essence it is just an arbitrary tie-breaking procedure, so in practice oscillations do not alter the average performance.

A Hopfield network of N neurons can safely store and recall approximately $0.14N$ patterns (with noiseless cues) (Amit *et al.*, 1985). However once the capacity of the Hopfield network is exceeded there is catastrophic failure, and the network's only stable states are spurious states, unrelated to the original patterns (Hopfield, 1982). By viewing recall as probabilistic inference we obtain update rules which are similar to Hopfield's yet have better performance and do not suffer from catastrophic failure (MacKay, 1991). We now explore several different forms of probabilistic decoding (while keeping the encoding of the original Hopfield network).

4.1.1 Belief propagation in traditional Hopfield networks

As our first approach, we apply loopy belief propagation to the original Hopfield network. We use probabilities to represent our belief in the state of the initial cue c , which is more complex than the original formulation of a Hopfield network with only binary states, but allows more expressiveness. For example half of x might be known to have zero errors, while the other half is completely unknown; with a probabilistic formulation we are able to express the different certainties, while the Hopfield network would require sampling from or thresholding those probabilities.

Since the method of generating z is known we can calculate the posterior probability $\Pr(x \mid c, z)$ using Bayes' rule:

$$\Pr(x \mid c, z) \propto \Pr(x \mid c) \Pr(z \mid x). \quad (4.5)$$

4.1 The traditional Hopfield network

This joint probability is complicated and impractical to handle directly, so we make a simplifying assumption that the storage (weights) z are independent of each other. That this simplifying assumption discards information can be illustrated as follows. If after storing R patterns we find that $z_{ij} = R$ and $z_{jk} = R$ then we know for certain that for all the patterns stored $x_i = x_j$ and $x_j = x_k$. We can then deduce that for all patterns $x_i = x_k$ and hence $z_{ik} = R$. In making an independence assumption we lose the ability to reason about the correlation between multiple weights, but it allows us to calculate the marginals for every bit $\Pr(x_n | z)$ using loopy belief propagation. The simple storage rule results in a two-stage message-passing algorithm.

The algorithm will be described in terms of the state of three tables, f , g and d . The initial cue contains a binary vector with some proportion of the bits flipped. This gives us the probability that $\Pr(x_n = 1) = c_n$, where c_n is related to the probability of a bit flip. Observing a single z_{ij} provides evidence for the value $x_i \cdot x_j$ that was added when the pattern was stored. This evidence is stored in d_{ij} and needs to be calculated only once:

$$\begin{bmatrix} \bar{d}_{ij} \\ d_{ij} \end{bmatrix} = \begin{bmatrix} \Pr(z^{R-1} = z_{ij} + 1) \\ \Pr(z^{R-1} = z_{ij} - 1) \end{bmatrix}. \quad (4.6)$$

We start by initialising $g_{ij} = 1$ and then calculating

$$\begin{bmatrix} \bar{f}_{ij} \\ f_{ij} \end{bmatrix} = \begin{bmatrix} \bar{c}_i \prod_{j' \neq j} \bar{g}_{ij'} \\ c_i \prod_{j' \neq j} g_{ij'} \end{bmatrix}. \quad (4.7)$$

Now that we have an estimate for f , we can calculate a better estimate for g :

$$\begin{bmatrix} \bar{g}_{ij} \\ g_{ij} \end{bmatrix} = \begin{bmatrix} \bar{f}_{ji} d_{ij} + f_{ji} \bar{d}_{ij} \\ f_{ji} \bar{d}_{ij} + \bar{f}_{ji} d_{ij} \end{bmatrix}. \quad (4.8)$$

We can now repeatedly apply equations 4.7 and 4.8. Once the updates have converged (or a fixed number of iterations has been reached) the final approximate bit-wise marginals are read out:

$$\begin{bmatrix} \Pr(x_n = -1 | z) \\ \Pr(x_n = +1 | z) \end{bmatrix} \propto \begin{bmatrix} \bar{c}_n \prod_{n'} \bar{g}_{nn'} \\ c_n \prod_{n'} g_{nn'} \end{bmatrix}. \quad (4.9)$$

4. THE HOPFIELD NETWORK

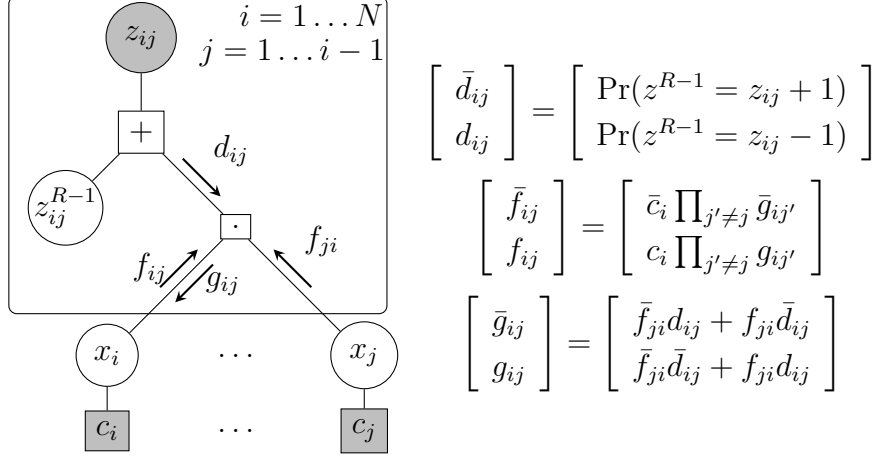


Figure 4.2: The update equations for loopy belief propagation in a Hopfield network.

4.1.2 Coordinate descent on the joint probability

A related method of decoding performs coordinate descent on the posterior (see Lengyel *et al.* (2005), particularly the supplementary material). To update a single x_n we consider the probability of it being on or off given the current, most likely setting of the rest of x . We then set the current x_n to its most likely state and consider another variable $x_{n'}$. Since at each stage the joint probability of the recall either remains the same or it increases, then we are guaranteed to converge to a solution.

The factor graph is the same as for belief propagation in figure 4.2 and the update equations would be identical to those given in equations 4.6–4.8, but we now binarize the values in equation 4.7 and use a Gaussian approximation in equation 4.6. The thresholding corresponds to maximum *a posteriori* decoding, and the Gaussian approximation lets us rework the update equations into a form that is linear in the log-odds. These simplifications can be written as follows:

$$\log \left(\frac{\Pr(x_i = +1 | z, x_{-i})}{\Pr(x_i = -1 | z, x_{-i})} \right) = \log \left(\frac{\Pr(x_i = +1)}{\Pr(x_i = -1)} \right) + \sum_j \log \left(\frac{\Pr(z = z_{ij} | x_i = +1, x_{-i})}{\Pr(z = z_{ij} | x_i = -1, x_{-i})} \right). \quad (4.10)$$

The prior belief is then given by:

$$\log \left(\frac{\Pr(x_i = +1)}{\Pr(x_i = -1)} \right) = \log \left(\frac{c_i}{1 - c_i} \right). \quad (4.11)$$

The simplified evidence from observing z_{ij} and x_j :

$$\begin{aligned} \log \left(\frac{\Pr(z = z_{ij} | x_i = +1, x_{-i})}{\Pr(z = z_{ij} | x_i = -1, x_{-i})} \right) &= \log \left(\frac{\exp \left(-\frac{(z_{ij} - x_j)^2}{2(R-1)} \right)}{\exp \left(-\frac{(z_{ij} + x_j)^2}{2(R-1)} \right)} \right) \\ &= 2 \frac{z_{ij} x_j}{R-1}. \end{aligned} \quad (4.12)$$

A single x_i is then updated according to:

$$x_i = \Theta \left(\log \left(\frac{c_i}{\bar{c}_i} \right) + \sum_{j \neq i} 2 \frac{z_{ij} x_j}{R-1} \right). \quad (4.13)$$

Using the exact Binomial distribution can sometimes result in performance that is worse than using the Gaussian approximation. This occurs when $z_{ij} = R$ or $z_{ij} = -R$ (which typically implies that only a small number of patterns have been stored). If either x_i or x_j are incorrect in the initial cue and we condition on the incorrect variable then observing z_{ij} gives infinite evidence for the incorrect value of the other variable. This is a side-effect of conditioning on noisy observations, yet treating the observations as noiseless. In section 4.1.4 we show how to correct this flaw.

4.1.3 Maximum entropy decoding

The final method we consider is known as Maximum entropy decoding (MacKay, 1991). This method ensures that the joint posterior is maximally uncertain yet still satisfies the constraints imposed by the known covariance of the storage. In this case we need to find a distribution $\Pr(x)$ that satisfies the following constraints:

$$\sum_x x_i \cdot x_j \Pr(x) = \frac{z_{ij}}{R}. \quad (4.14)$$

Using Lagrange multipliers, the distribution with maximal entropy then takes the form:

$$\Pr(x | z) \propto \exp \left(\sum_{i,j} w_{ij} x_i x_j \right). \quad (4.15)$$

Unfortunately setting w_{ij} directly is hard. It is possible to find the correct w_{ij} by training a Boltzmann machine (Ackley *et al.*, 1985), but we shall instead follow the approach of MacKay (1991) and explicitly find the marginal probabilities of a single

4. THE HOPFIELD NETWORK

variable given only the covariances between that variable and all other variables (but ignoring the covariances between the other variables). As we are storing completely random binary vectors we can use the formula given in MacKay (1991):

$$w_{ij} = \tanh^{-1} \left(\frac{z_{ij}}{R} \right). \quad (4.16)$$

So far we have not included the prior which will provide a bias for x . Writing the prior as:

$$\Pr(x_n | \text{cue}) = \sqrt{c_n^{1+x_n} (1 - c_n)^{1-x_n}}, \quad (4.17)$$

which we can rewrite in log-prob form as:

$$\log(\Pr(x_n | \text{cue})) = \frac{x_n}{2} \log \left(\frac{c_n}{1 - c_n} \right) + \text{constant}. \quad (4.18)$$

The constant can be ignored as the entire distribution will need to be normalised. This gives the final form of the maximum entropy distribution.

$$\Pr(x | c, z) \propto \exp \left(\sum_n w_n x_n + \sum_{i,j} w_{ij} x_i x_j \right), \quad (4.19)$$

where:

$$w_n = \frac{1}{2} \log \left(\frac{c_n}{1 - c_n} \right), \quad (4.20)$$

and we have used our usual notation that $c_n = \Pr(x_n = +1)$.

Once we have the joint probability, we are again able to perform coordinate descent by calculating the posterior probability for a state and setting it to the most likely value. Considering the log-odds for a single x_i given the state of all other $x_{\setminus i}$:

$$\log \left(\frac{\Pr(x_i = +1 | z, x_{\setminus i})}{\Pr(x_i = -1 | z, x_{\setminus i})} \right) = 2 \sum_{j \neq i} w_{ij} x_j + 2w_i. \quad (4.21)$$

We can then update x_i according to:

$$x_i = \Theta \left(\log \left(\frac{c_i}{\bar{c}_i} \right) + \sum_{j \neq i} 2 \tanh^{-1} \left(\frac{z_{ij} x_j}{R} \right) \right). \quad (4.22)$$

4.1.4 Uncertainty in the cue

Both the coordinate descent and the maximum entropy decoding algorithms suffer from a slight model mismatch, in that they assume the rest of the bits in x are correct when trying to predict x_i . In fact a significant proportion of the bits in x are incorrect.

When conditioning on x_j , we aren't conditioning on the correct bit, instead we are conditioning on a bit that may have been flipped. Mathematically this is equivalent to conditioning on the XOR of the correct bit and a bit indicating whether a flip has occurred or not. This is very similar to the parity codes which we cover in the next chapter; in section 5.2.3 we show that in the log-odds domain the evidence from a parity bit is limited by the most uncertain bit. In the Hopfield network this means that the evidence from observing x_j and z_{ij} should never be greater than the evidence for x_j having its observed value. We ensure that no evidence from a single bit is greater than the uncertainty in the bit, by defining the symmetric threshold function:

$$\kappa_\nu(x) = \begin{cases} -\nu & \text{if } x < -\nu \\ x & \text{if } -\nu \leq x \leq \nu \\ \nu & \text{if } x > \nu \end{cases} . \quad (4.23)$$

This is then applied to equation 4.12 to give:

$$x_i = \Theta \left(l_i + \sum_{j \neq i} \kappa_{l_j} \left(2 \frac{z_{ij} x_j}{R-1} \right) \right), \quad (4.24)$$

where $l_i = \log \left(\frac{c_i}{\bar{c}_i} \right)$. Adding in cue uncertainty to equation 4.22 gives rise to the following update equations:

$$x_i = \Theta \left(l_i + \sum_{j \neq i} \kappa_{l_j} \left(2 \tanh^{-1} \left(\frac{z_{ij} x_j}{R} \right) \right) \right). \quad (4.25)$$

Adding this uncertainty allows better initial performance when only a few patterns have been stored. Ideally this strength could be changed after each step of decoding has been performed, to reflect the fact that the probability of bit error in x has now changed, although we did not investigate this further. As an aside we found that changing the maximum strength to an artificially low number improved the empirical performance of an overloaded Hopfield network (although this came at the cost of decreased maximal performance for the critically-loaded network).

4. THE HOPFIELD NETWORK

4.1.5 Results

Comparing equations 4.24 and 4.25, we see that two very different approaches have led to nearly identical update rules. One might expect the non-linearity of \tanh^{-1} to improve recall, however in practice the limit on the message strength from the cue uncertainty ensures that the contribution from \tanh^{-1} is essentially linear. There is no noticeable difference in performance between the two methods (as shown in figure 4.3), and the methods clearly outperform both belief propagation and the traditional Hopfield approach.

Our results are broadly consistent with the literature. The original Hopfield network can store $0.14N$ patterns which MacKay (2003) has characterised as 0.24 bits per integer. However this work used cues with very low levels of noise (around 1% bit-flips) and according to our measure of capacity, once the information from the cue is subtracted out this performance is not optimal. In general we found that the network was most informative when the cue had approximately 20% noise (in figure 4.4(a)), and the capacity of the traditional Hopfield network was **0.14** bits per integer. Using the probabilistic methods above, the performance improved to approximately **0.17** bits per integer (as shown by the grey line in figure 4.4(b)).

Figure 4.3(b) suggests that the maximum total information recalled for both coordinate descent and maximum entropy occurs approximately when the number of patterns stored, R , is $0.14N$. The extra levels of noise in the cue mean that the traditional Hopfield network can now store only $R = 0.10N$ patterns for maximal recall under our performance measure.

Belief propagation does not perform as well as either the coordinate descent method or the maximum entropy method. We believe that this is due to the loopiness of the factor graph and the dense encoding signal (in other associative memory tasks with *sparse* encoding functions we examined the performance of coordinate descent and found it inferior to belief propagation).

4.1.6 Related Linear networks

There is a rich literature on associative memories which use addition to store representations of patterns (or pairs of patterns in the heteroassociative case). The covariance

4.1 The traditional Hopfield network

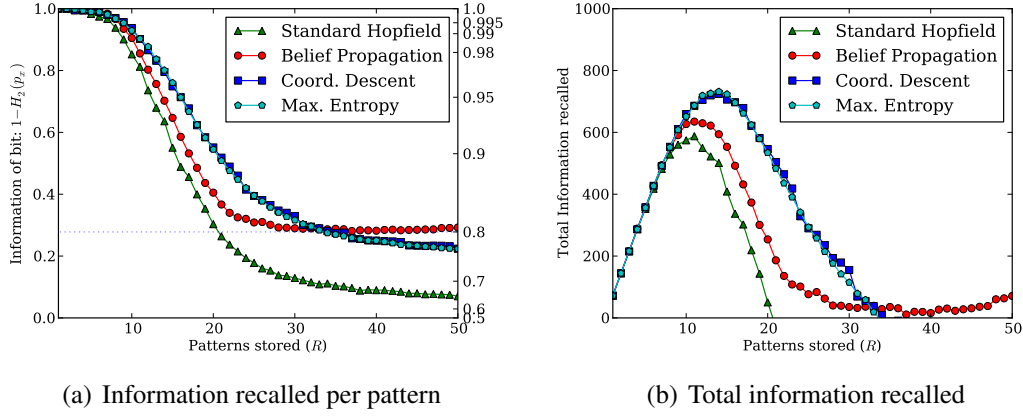
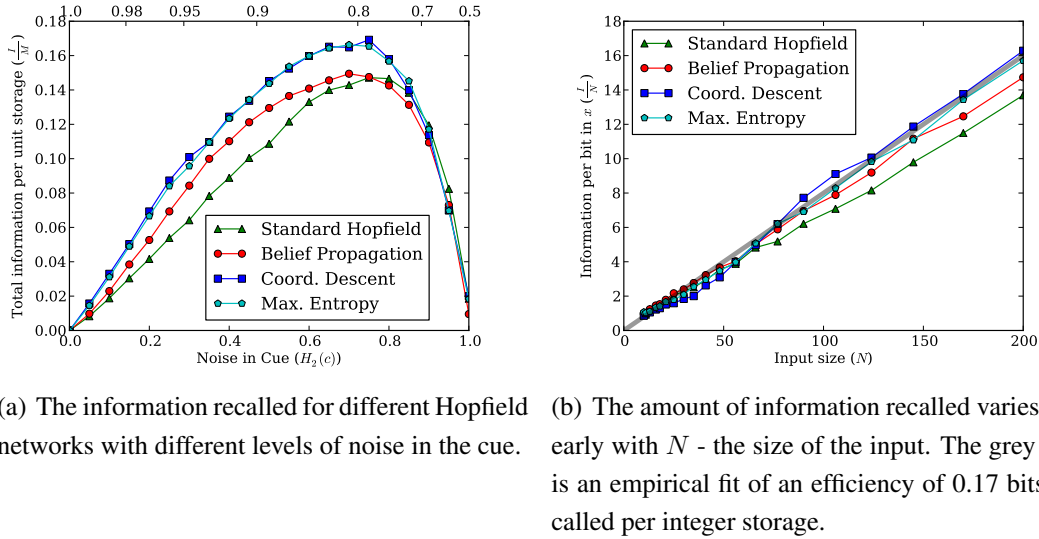


Figure 4.3: The recall performance of several different variants of a Hopfield network. Here we stored up to $R = 50$ patterns of $N = 100$ bits each in a storage of $M = 4950$ integer weights.



(a) The information recalled for different Hopfield networks with different levels of noise in the cue. (b) The amount of information recalled varies linearly with N - the size of the input. The grey line is an empirical fit of an efficiency of 0.17 bits recalled per integer storage.

Figure 4.4: Hopfield network's recall performance for increasing network size and increasing levels of noise in the initial cue.

4. THE HOPFIELD NETWORK

update rule (Sejnowski, 1977) is a generalization of the Hopfield update rule. The covariance rule gives the strength of terms that should be added when the patterns are not random binary vectors. The central limit theorem implies that if many associations are stored in a network and each association adds to a value in storage then the storage will have a Gaussian distribution. The central limit theorem only applies after many terms have been added together, which is very likely to happen in an encoding which is dense. Sparse encodings are unlikely to have sufficiently many non-zero terms for the Gaussian approximation to be accurate. As the work in previous sections show, the Gaussian approximation leads to linear inference rules which are optimal (Lengyel *et al.*, 2005). To ensure the highest capacity the signal-to-noise ratio should be maximised as this gives the best separability between the conditional distributions for the output bit being on or off (Dayan & Willshaw, 1991).

For the heteroassociative case in which pairs of patterns (x, y) are memorised there are four possible values for the synaptic updates, corresponding to all possible settings of x_i and y_j :

	$\neg y$	y
$\neg x$	α	β
x	γ	δ

If the learning rule is unbiased then the mean synaptic value is unchanged, and this is a necessary requirement for an optimal learning rule (Dayan & Willshaw, 1991). However it is possible to recover good performance (in the sense that the number of recallable patterns is linear in the number of neurons) through synaptic normalisation (Chechik *et al.*, 2001). Letting $p_x = \Pr(x = 1)$ and $\bar{p}_x = \Pr(x \neq 1)$ with similar conventions for y , then the covariance rule is given by (Dayan & Willshaw, 1991):

$$\begin{aligned} \alpha &= +\bar{p}_x\bar{p}_y & \beta &= -\bar{p}_xp_y \\ \gamma &= -p_x\bar{p}_y & \delta &= +p_xp_y. \end{aligned}$$

The signal-to-noise ratio is invariant under rescaling of the update equations; with random binary vectors for the input and output ($p_x = p_y = 0.5$) one can rescale the update equations and recover the Hopfield update rule.

Signal-to-noise analysis can also be used to provide insight into the effects on capacity under a range of conditions that are known to be biologically relevant (Sterratt & Willshaw, 2008). Sterratt & Willshaw (2008) look at the effects of attenuation due

4.1 The traditional Hopfield network

to branched versus unbranched dendritic trees and find a surprisingly low reduction in the capacity. They also analyse the effect of probabilistic transmission on the signal-to-noise ratio, which is particularly relevant as it is known that synapses are unreliable in the release of synaptic vesicles (Bolshakov *et al.*, 1997).

On a technical note, it is possible to store arbitrary amounts of information in a real value (as well as an unbounded integer). As an example, one could encode all four counts of the conditions that have occurred in a single real value by assigning unrelated irrational numbers to each of α , β , γ and δ . With this encoding there is only a single solution for the equation $a\alpha + b\beta + c\gamma + d\delta = 0$ with integer coefficients. The proper Bayesian approach would then take into account the fact that the four encoded integers provide evidence for the probability of each condition. However this dissertation is focused on methods that are inspired by biological neurons and will abide by a gentleman's agreement not to use the full representational ability of integers and real numbers in unrealistic ways.

Even if encoding a number of integers in a single real number is possible but unrealistic it is still interesting on how best to decode the output given a number of statistics about the stored patterns. Knoblauch (2010a) looks at the optimal Bayesian decoding in a heteroassociative memory using several statistics for each pair of variables (x_i, y_j) . Not only is the mean activation for each binary variable stored:

$$z_i = \sum_r x_i, \quad (4.26)$$

$$z_j = \sum_r y_j, \quad (4.27)$$

but also the statistics of individual conditions are stored:

$$\begin{aligned} z_{ij}^{00} &= \sum_r (1 - x_i^r)(1 - y_j^r) & z_{ij}^{01} &= \sum_r (1 - x_i^r)y_j^r \\ z_{ij}^{10} &= \sum_r x_i^r(1 - y_j^r) & z_{ij}^{11} &= \sum_r x_i^r y_j^r. \end{aligned}$$

Using these statistics Knoblauch (2010a) is able to find a non-linear rule which performs maximum-likelihood decoding in a single step, with an efficiency comparable to traditional Hopfield networks. There is a related paper which also shows that the computations can be achieved with very little loss in efficiency if one is restricted to using statistics with low precision (Knoblauch, 2010b).

4. THE HOPFIELD NETWORK

4.2 Improving on pairwise encoding

In a Hopfield network, recall fails when there is too much noise from other patterns in the storage. This suggests that better performance could be obtained if the encoding function $f(x)$ were sparse. In the next section we create a sparse signal by testing for the equality of many variables. By using functions of many variables we also hope that observing a particular z_m will give us information about many bits in x . We also briefly consider a completely different approach and extend the Bloom filter of the previous chapter, creating an associative memory out of a counting Bloom filter.

4.2.1 The neurally-inspired Hopfield network

Extending the Hopfield encoding function by considering the multiplication of several variables still creates a dense encoding. As a first step towards a sparse encoding function we suggest the “symmetric AND” function \otimes defined as follows:

$$x \otimes y = \begin{cases} x & \text{if } x = y \\ 0 & \text{otherwise.} \end{cases} \quad (4.28)$$

This has the value of generalising to multiple variables:

$$\otimes(x_1, x_2, \dots, x_a) = \begin{cases} -1 & \text{if all } x_n = -1 \\ +1 & \text{if all } x_n = +1 \\ 0 & \text{otherwise.} \end{cases} \quad (4.29)$$

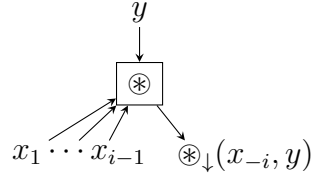
We also randomly negate the inputs for each f_m , as this improves the independence of each f_m from other f_m ’s (as a motivating example, the storing the all-ones pattern will result in a dense encoding unless we include random negations). Note that the symmetric-AND of two variables is different from the Hopfield encoding in the previous section.

The storage rule is still addition so that storing the r^{th} pattern x in a neurally-inspired Hopfield network is done by calculating $z_m^r = z_m^{r-1} + f_m(x)$. An overview of the process is shown in figure 4.5. The symmetric-AND function is also interesting as one can remove a pattern by storing $-x$. While the symmetric-AND has several attractive computational properties we have no guarantee that this is the best encoding function and leave the examination of other suitable sparse encoding functions as a direction for future work.

4.2 Improving on pairwise encoding

By calling the network presented here a neurally-inspired Hopfield network, we do not wish to convey that the original Hopfield network was not inspired by biological neurons. We merely wish to emphasize that our network relies on the combination of random connectivity with principled statistical inference to perform recall. The network also has the same type of storage as a Hopfield network (i.e. integers), and similar message-passing techniques are used to perform recall.

The message-passing consists of only two phases, an up phase and a down phase. The down phase is given by:



$$\circledast_{\downarrow}(x_{-i}, y) = \begin{bmatrix} y^{\circ} + (y^{-} - y^{\circ}) \left(\prod_{j \neq i} \bar{x}_j \right) \\ y^{\circ} + (y^{+} - y^{\circ}) \left(\prod_{j \neq i} x_j \right) \end{bmatrix} \quad (4.30)$$

where $[y^{-}, y^{\circ}, y^{+}]$ represents the messages for $y = [-1, 0, +1]$ respectively:

$$\begin{bmatrix} y^{-} \\ y^{\circ} \\ y^{+} \end{bmatrix} = \begin{bmatrix} \Pr(z^{R-1} = z_m + 1) \\ \Pr(z^{R-1} = z_m) \\ \Pr(z^{R-1} = z_m - 1) \end{bmatrix} \quad (4.31)$$

The up phase collects all-but-one of the messages coming down to a particular variable together with the prior and combines them to give the message for the excluded connection:

$$m_{n \rightarrow m}(x_n) \propto \prod_{m' \neq m} \circledast_{\downarrow m' \rightarrow n}(x_{-n}, y_m) \quad (4.32)$$

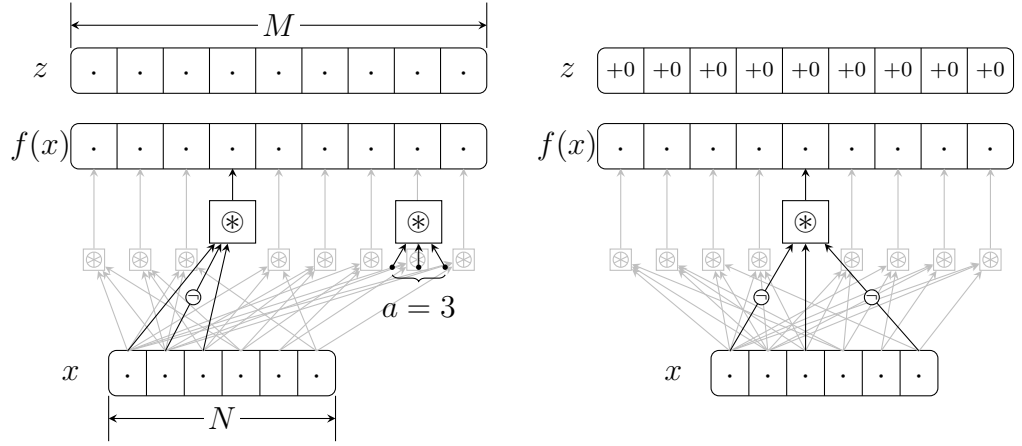
These two phases are alternated until convergence occurs (or a fixed number of iterations is exceeded). The final marginals are read out using:

$$\Pr(x_n | z, c) \propto c_n \cdot \prod_m \circledast_{\downarrow m \downarrow n} \quad (4.33)$$

and these can be thresholded using maximum likelihood to give the final binary answer.

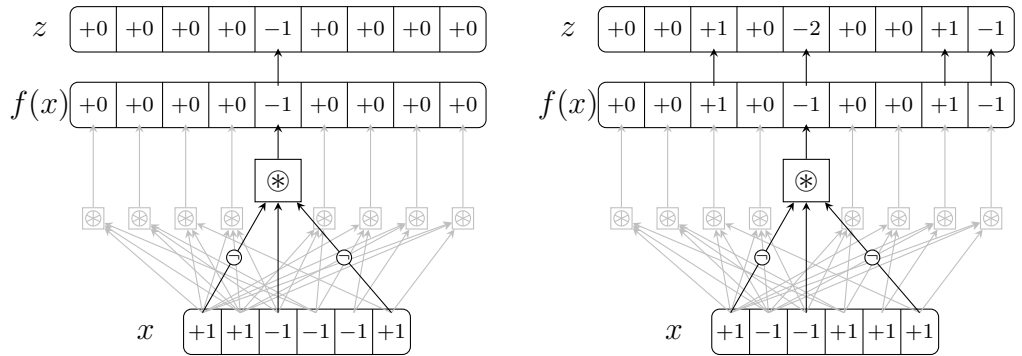
The recall performance for different levels of noise are shown in figure 4.6. We obtained this figure by storing many patterns of size $N = 100$ in storage of size

4. THE HOPFIELD NETWORK



(a) The structure of a neurally-inspired Hopfield network.

(b) z is initialised to the zero vector.



(c) To store a pattern x , we add $f(x)$ to z .

(d) To store another pattern we again add $f(x)$ to z .

Figure 4.5: Storage in the neurally-inspired Hopfield network. The encoding is done through the symmetric-AND function which returns 0 if any of the inputs differ, if all the inputs are the same it returns the unique input. Note that the encoding is sparse compared to the original Hopfield network.

$M = 4950$. For each level of noise we tested multiple values of a and different numbers of patterns to store and then plot only the best performance. We can see that optimal performance occurs for cues with a noise level of around 10%. For comparative purposes we plot the performance of the Hopfield networks from the previous section in light grey. We can see that the sparse encoding function allows a large improvement in the recall efficiency of the network.

We now briefly examine a different direction, using the integer storage for a counting Bloom filter and turning it into an associative memory.

4.2.2 The Neurally-inspired Counting Bloom filter

Another way to use the integer storage in z is to create a counting Bloom filter (Fan *et al.*, 2000) using techniques from the previous chapter. Rather than measuring whether $h_m(x)$ has ever returned true, this Bloom filter counts the *number* of times h_m has returned true. Counting Bloom filters perform all of the operations of a normal Bloom filter and are also able to remove items from the set they represent. To remove an item x^* from the counting Bloom filter one subtracts $h(x^*)$ from z_m . In practice z_m remains small and only a handful of bits are needed to store each count.

When querying whether \hat{x} is an element of the counting Bloom filter one evaluates $h(\hat{x})$ and if any $h_m(\hat{x})$ returns true, yet z_m is zero, then \hat{x} is not an element of the set. Since testing for membership only involves testing against zero the probability of a false positive is the same as in the binary case. However one can use the integers in z to improve the inference when performing the associative memory task. Loosely speaking if z_m is large then it is more likely that the recall involves an x for which $h_m(x)$ is true. Surprisingly this effect is rather small - as one can see in figure 4.6 - increasing the efficiency of the associative memory from 0.36 bits recalled per bit of storage to **0.45** bits recalled per *integer* of storage. It appears therefore that most of the information still comes from observing $z_m = 0$ which implies with certainty that the boolean hash function $h_m(x)$ has never returned true. In keeping with this finding we also found that the optimal parameters were unchanged from the binary case. As a result we only mention this network in passing, rather than dedicate an entire section to it.

4. THE HOPFIELD NETWORK

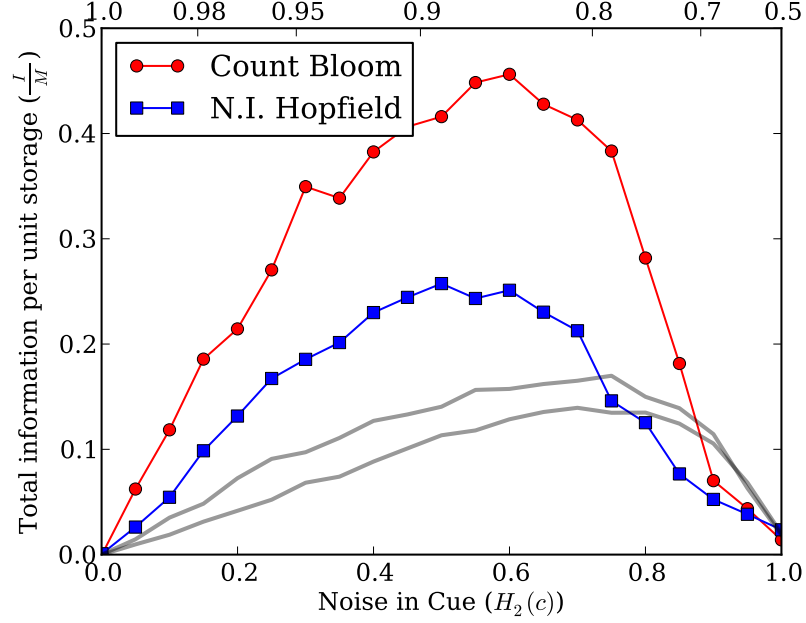


Figure 4.6: Recall performance improves with a sparse encoding function. For both the counting Bloom filter and the neurally-inspired Hopfield network the sparse encoding is achieved through the conjunction of many variables. The performance of the previous section’s Hopfield networks are shown in light grey for comparative purposes. The ideal level of cue noise has now decreased to 0.1, suggesting a cleaner signal is required to perform inference on the conjunction of many variables.

In light of the small increase in performance when creating an associative memory out of a counting Bloom filter, one might want to create a palimpsest memory instead. Storing a pattern x in this palimpsest memory can be done by setting z_m to R whenever $h_m(x)$ returns true, and decrementing all other non-zero entries in z . In this manner a pattern will be forgotten after another R patterns have been stored. If we consider a non-zero entry in z to be equivalent to true in the binary case, then the memory is completely identical to the binary associative memory from 3.

4.3 Conclusions

In this chapter we cast the Hopfield network into a similar form as the associative memories of the preceding chapter. The Hopfield network has a single integer in each storage element instead of a bit, and measures the correlation between two bits in x . To obtain the representation for x each pair of bits is multiplied together to create a dense representation which is then added to z . Examining a known associative memory provides us with a benchmark of performance for other associative memories.

Recall in the traditional Hopfield network use rules that are guaranteed to converge (Hertz *et al.*, 1991), yet these rules cannot be interpreted as performing inference. We then explored three different recall methods which can be interpreted as performing inference: belief propagation, coordinate descent and maximum entropy. All three methods of recall perform better than recall in the traditional Hopfield network. The traditional Hopfield network is able to provide **0.14** bits of information per unit of storage using our performance metric. This can be improved to approximately **0.17** bits by performing either the coordinate-descent method or the maximum-entropy method on the posterior. The biggest difference between traditional recall and probabilistic inference is the prior, which provides a constant bias during recall. We also showed that the update equations for maximum entropy and coordinate descent are very similar. This is especially true when fairly noisy cues are used, as the non-linearity in the maximum entropy method is effectively ignored, and the final update equations are essentially identical to the linear coordinate-descent method.

We then looked to improve on the performance of the Hopfield network by generalizing to a network which encodes a pattern based on the equality of several variables. This is in contrast to the Hopfield network which only considers pairs of variables to encode a pattern. This sparsifies the storage signal and allows the successful recall of more patterns. It also ensures high levels of connectivity so that inference of a bit in x relies on many bits in z . We called this network the neurally-inspired Hopfield network and showed that it had an efficiency of **0.25** bits per integer storage. This compares favorably to the **0.17** bits per integer efficiency of the probabilistic decoding of a Hopfield network.

The single integer in each element of storage can also be used to create a counting Bloom filter. This allows elements to be added and removed from the Bloom filter. The

4. THE HOPFIELD NETWORK

counts themselves also provide information when the counting Bloom filter is used as an associative memory. We found the performance increased from **0.36** bits of recall per bit of storage for a normal Bloom filter, to **0.45** bits of recall per integer of storage for a counting Bloom filter.

This chapter has provided evidence that:

- **Principled** recall is better than an ad-hoc method of recall. This means that one can consider different methods of encoding, and the optimal recall algorithm can be mechanically derived from Bayes' theorem.
- **Sparse** representations appear to superimpose far better than dense representations. This provides weak guidance into what sort of encodings will have good capacities.

The palimpsest memory in the previous chapter overwrote bits independently, and was relatively inefficient as a result. In this chapter we have avoided overwriting bits independently by using integer storage which allows patterns to be superimposed through addition. In the next chapter we will explore an alternative method to avoid overwriting bits independently. The next chapter will overwrite bits in a dependent fashion. It does so by grouping bits together to form small populations which are overwritten as a whole. The representation used for each population is inspired by research from error-correcting codes.

CHAPTER 5

Associative Memory from Error-Correcting Codes

This chapter creates an associative memory that is very efficient for low-noise cues, yet still tolerates hardware failure. We do this by borrowing concepts from error-correcting codes to create an associative memory. We make the key assumption that the noisiness of the cue is known in advance and we know how many patterns in total are to be stored.

The error-correcting codes we examine are a form of low-density generator matrix (LDGM) codes and are bad codes in the sense that reducing their probability of error to zero requires that the rate of the code goes to zero as well (Richardson & Urbanke, 2008), however they are quick to encode and decode. Their simplicity also makes them resistant to hardware failure. If we only care about creating efficient associative memories then we'll see that reducing the error rate to zero is non-optimal, and the "bad"ness of these codes is irrelevant.

We first examine error-correction in the case where the storage is faulty, and there is no initial cue. This is the standard setting for error correcting codes. In the second section we then assume that an initial cue is provided for x , and that there is no noise in our storage, both of which reduce the required information in z and we examine the situation where the storage is smaller than the item we're trying to recover (i.e. $M < N$). Taken together x and z can be thought of a systematic code (with different noise levels), and we call this cued error-correction.

We then group many such error-correcting codes together to form an associative memory. Each error-correcting code is split into two parts, a low-connectivity¹ part

¹Here the term "connectivity" is used to refer to the average number of bits in x that are used to encode a single bit in z . Typically low-connectivity will refer to 1-2 input bits per encoding function. High-connectivity will typically refer to an encoding function involving 3-40 input bits.

5. ASSOCIATIVE MEMORY FROM ERROR-CORRECTING CODES

that is easy to infer, and a high-connectivity part that reduces the error rate. When performing recall from an initial cue we first use the predictions from the low-connectivity part to identify the correct memory and then perform message-passing on the corresponding high-connectivity part. Splitting the code into two allows us to give the necessary scaling for the low-connectivity part if we want the same probability of error as more patterns are stored.

5.1 Introduction to Error-Correcting Codes

An error-correcting code is a code that adds redundancy into data in such a way that if the data is corrupted by a small amount of noise it is generally still possible to recover the original data. Many things can be thought of as error-correcting codes e.g. the English language has some redundancy, so that most spelling and grammar mistakes do not render a sentence meaningless, or change the meaning completely.

Error-correcting codes are designed to achieve reliable communication over a noisy *channel*. There are many examples of noisy channels, e.g. bits could be noisily read from a hard drive, or there could be electromagnetic interference in a satellite transmission. The channel is viewed as inherently unreliable and will not reproduce the original message exactly. To achieve reliable communication we can transmit a longer message that has some redundancy which will allow us to infer the interference and recover the original message. Moving from the original message to the redundant representation is known as encoding the error-correcting code, while decoding the redundant message will (hopefully) result in the original message.

The two most important parameters of any error-correcting code are the *rate* of the code and the *probability of error*. The rate of a code is the ratio of the length of the encoded message to the length of the decoded message. If a 100 bit message is encoded into 200 bits then the code has a rate of 0.5. For any finite-length message there is a non-zero probability of error (when the decoded message is not equal to the original message).

Shannon (2001) showed that non-zero rate codes can have zero probability of error if one considers a sequence of codes with increasing block length. If the block length is scaled linearly $O(N)$ then the standard deviation of the number of errors that the channel introduces only scales as $O(\sqrt{N})$. This means that the proportion of errors

that the channel introduces becomes increasingly precise. In the limit of an infinitely-long message the proportion of errors will be exact and if a code can successfully correct that proportion of errors then the code will have zero probability of error.

Shannon (2001) also proved the mathematical limits on the rate of a code for a channel with given noise characteristics. This limit is known as the *capacity* of the channel.

Error-correcting codes are generally used in a different setting from the one we consider in the rest of the chapter. Typically the code is designed such that the encoding (or decoding) is achieved by a single piece of hardware and then the encoded message (or corrected message) is sent to its destination and the hardware can then immediately begin processing more data. In contrast we will allocate specific hardware to a particular pattern and group the pattern and the hardware together for an extended length of time. When another pattern is stored in our associative memory we will allocate extra hardware that will exclusively represent that particular pattern.

Repetition codes (MacKay, 2003) are the simplest codes against which we benchmark. To encode a pattern we simply make an odd number of copies of that pattern. The encoding can be decoded optimally though a simple majority calculation for each individual bit.

Bose & Ray-Chaudhuri (1960) and Hocquenghem (1959) independently developed the same algorithm to encode and decode an error-correcting code. It is known as the BCH algorithm after the authors surnames and found widespread application (e.g. in satellite communication, CD-ROM encodings and DVD encodings) due to a particularly simple decoding algorithm. More modern codes are now replacing it such as LDPC codes (MacKay, 2003) which are close to the Shannon limit, but it is more appropriate to compare the simple error-correcting codes presented here with BCH codes. Reed-Muller codes (RM codes) (Van Lint, 1999) are also used as a benchmark. Both RM codes and BCH codes have algebraic origins and a detailed description is beyond the scope of this dissertation.

5.2 Simple Error-Correcting Codes

For a neurally-inspired error-correcting code we require the ability to allocate and de-allocate computational units to the code. This must not require changing any of the

5. ASSOCIATIVE MEMORY FROM ERROR-CORRECTING CODES

remaining computational units. We also would like the error-correcting capability to improve as more computational units are added to it. A low-density generator matrix code (LDGM) can achieve this (Richardson & Urbanke, 2008).

At a high level, we have an N -dimensional binary input vector x . We encode x into a larger M -dimensional binary output vector z using a different random function for each bit: $z_m = f_m(x)$ (see figures 5.1(a) and 5.1(b)). This is the encoding that can be stored in noisy long-term memory. When the memory is recalled there will be incorrect bits in our storage: \hat{z} , and for concreteness we will assume that roughly $p_z M$ bits are flipped (see figure 5.1(c)). We can now use the redundancy in our encoding to infer what the correct x is (see figure 5.1(d)). This task is impossible to perform on a finite code with zero probability of error (since it is possible that more bits will be flipped than we can deal with) and we would like to estimate the new probability of bit errors in x which we will denote p_x . Ideally we would like to have low redundancy $M = cN$, e.g. $c \in (1, \dots, 10)$, and much better reliability $p_x \ll p_z$.

There already exists a large literature on error-correcting codes. There are similar codes to the ones presented here which approach the Shannon limit (known as Low-Density Parity-Check codes (MacKay & Neal, 1997)). These codes are expensive to encode – $O(N^2)$ – but there are computational tricks which can reduce the encoding to linear complexity (Richardson & Urbanke, 2002). These variations are known as repeat-accumulate codes or staircase codes (MacKay, 2003). Unfortunately they are fragile to hardware failure as the removal of any of the parity bits will break the encoding. A similar solution is the concatenated LDGM code (Oenning & Moon, 2001), which requires low connectivity and has linear encoding time, but requires two distinct layers of computation. This also makes the concatenated LDGM code sensitive to hardware loss as the loss of a unit in the first layer of computation will cause the failure of many units in the second layer.

It is possible that there exists a clever encoding (in which units depend on one another, encode in linear time and yet are not sensitive to hardware loss), however we have not been able to find such a scheme and will instead focus on the simpler solution of calculating the bits in isolation. An encoding scheme in which each bit is calculated independently of all others ensures the solution will be robust to hardware failure.

5.2.1 Encoding

The codes in this chapter all use a simple encoding scheme, where each output bit is computed in isolation of the other output bits:

$$z_m = f_m(x). \quad (5.1)$$

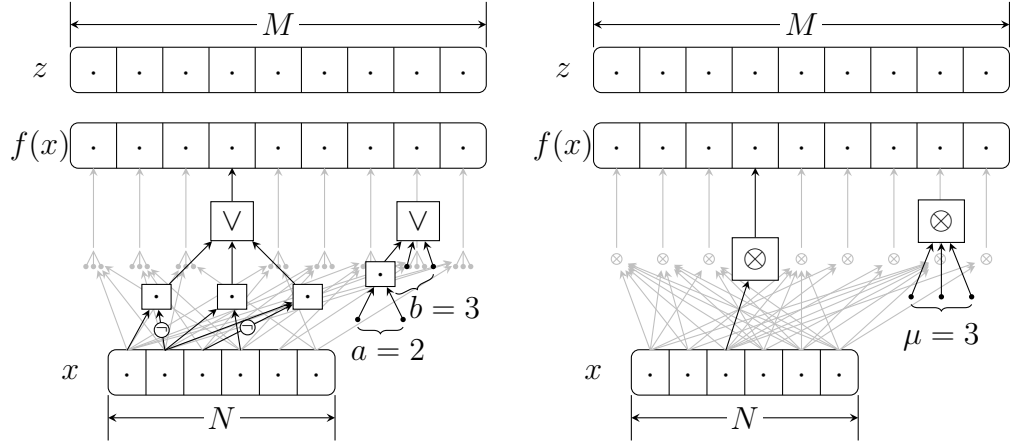
Ideally we would like each z_m to be as independent of each other as possible, since highly-correlated bits do not contain as much information. Near-independence can be achieved by using random boolean functions which depend on many input bits. However we will see that decoding such functions is hard. For the results shown in this section we considered two different encoding functions: the sigma-pi function and the parity function. The sigma-pi function has already been used previously in this dissertation; although now its parameters are set so that the probability of returning true is approximately 50% (shown in figure 5.1(a)). From equation 2.8 on 16 if we are given the number of “AND”s a , the function will have $b \approx 2^a \cdot \ln 2$ “OR”s. The parity function is shown in figure 5.1(b) and can be constructed out of repeated use of the “XOR” function:

$$f(x) = x_2 \otimes x_9 \otimes x_7. \quad (5.2)$$

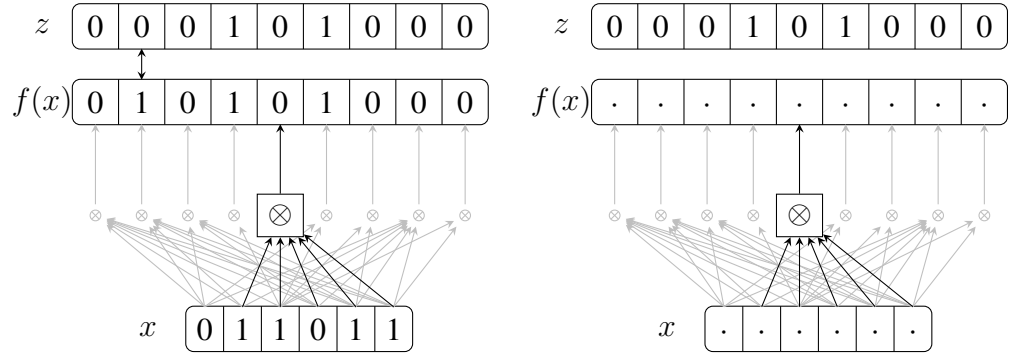
It is difficult to infer the parity of many variables, each of which is a little uncertain, since a single error gives the incorrect answer. If we hope to get good results then we must include the parity of single variables (which are easy to infer) with the parity of many variables (which produce good results). To achieve this we force each computational unit to have a connectivity of at least one and sample any additional connections from a geometric distribution with mean $\mu - 1$. Thus the expected connectivity is μ , and we plot results for different μ and different rates ($\frac{N}{M}$) of communication. Note that the code for $\mu = 1$ corresponds to a simple encoding of copying a randomly-chosen bit from the input into the output, and is a simple repetition code (although there will be different repetitions of individual bits).

The geometric distribution was chosen as it gave good empirical results. In section 5.2.3 we give some theoretical insight into the decoding behaviour from observing an increasing number of parity bits. This leads us to another distribution of connectivity which gives better performance at low rates, although further work in understanding this distribution is needed.

5. ASSOCIATIVE MEMORY FROM ERROR-CORRECTING CODES



(a) An error-correcting Sigma-Pi encoder. Note that the number of “AND”s and “OR”s have changed from the Bloom filter task. (b) A parity encoder. The mean connectivity of $f_m(x)$ is μ (but the particular connectivity for each bit is allowed to vary).

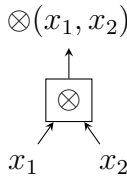


(c) To encode x we calculate $f(x)$ and store it in z . However there are storage errors, due to noisy hardware. (d) The task is to infer the most likely x given only the noisy \hat{z} . Loopy belief propagation is again used to perform the inference efficiently.

Figure 5.1: Creating a neurally-inspired error-correcting code.

5.2.2 Calculation of Messages

The messages for the sigma-pi codes have been given in section 3.1.4. We will instead focus on the messages for the parity code. The parity function can be built from repeated “XOR”ing, and has the useful property that “XOR” messages are symmetric in both directions. This can be seen from the fact that $x_1 \otimes x_2 = z$ has the same truth table as $x_1 \otimes x_2 \otimes z = 0$. The messages have the following form:



$$\otimes(x_1, x_2) = \begin{bmatrix} \bar{x}_1 \bar{x}_2 + x_1 x_2 \\ \bar{x}_1 x_2 + x_1 \bar{x}_2 \end{bmatrix} \quad (5.3)$$

Given that we have already settled on a cheap-and-cheerful encoding and accepted a probability of error it might seem incongruous that we are using a high-precision message-passing technique to solve the task. We believe it likely that there exists a low-precision decoder which performs nearly as well as the high-precision decoder; such low-precision encodings have been found for Low-Density Parity-Check codes (Richardson & Urbanke, 2001). However we leave this as a future research direction, and note that even if such a decoder existed we would still need the high-precision decoder to provide a benchmark of performance.

5.2.3 Theoretical Insight for the parity code

We can gain some theoretical insight into the performance of the parity code by manipulating and approximating the formulae. We start by converting everything into the log-odds domain.

$$l = \log \left(\frac{p}{1-p} \right) \quad (5.4)$$

The advantage of the log-odds domain is apparent when looking at the messages that need to be calculated from a variable (see equation 3.1 on page 35). The multiplications are turned into additions and there is no need for renormalisation:

$$\log \left(\frac{m_{n \rightarrow m}(x_n = 1)}{m_{n \rightarrow m}(x_n = 0)} \right) = \sum_{m' \neq m} \log \left(\frac{m_{m' \rightarrow n}(x_n = 1)}{m_{m' \rightarrow n}(x_n = 0)} \right) \quad (5.5)$$

5. ASSOCIATIVE MEMORY FROM ERROR-CORRECTING CODES

We will also frequently approximate the bound:

$$\log(e^x + e^y) \geq \max(x, y) \quad (5.6)$$

as being equal since this is a much easier form to manipulate. For clarity we consider different connectivities separately. We also consider only decoding $x = 0$, as this simplifies our notation. This means that the final log-odds should be negative for the correct decoding of a bit. We also assume that we are updating our belief in x after seeing a single bit in z and then discarding the bit. We start with the parity of a small number of bits and gradually increase the number of bits we consider. Estimating the probability of bit error by using this simplified sequential procedure is inherently conservative as it assumes there is only a single chance to learn from observing z_m . In the actual parallel implementation there is no particular order; high-connectivity parity bits will sometimes provide information about low-connectivity parity bits. (In our sequential procedure this would be classified as a failure of the low-parity bit.)

The simplest stage in the parity code copies a single bit from x into z . During decoding, the probability that the bit is now incorrect is p_z , therefore observing a single bit in z will either add or subtract:

$$l_z = \log \left(\frac{1 - p_z}{p_z} \right) \quad (5.7)$$

from the evidence for the appropriate bit in x . Since we are decoding $x = 0$, we will observe $z_m = 0$ and add l_z , with probability $1 - p_z$ (observing $z_m = 1$ will subtract l_z). This change in the log-odds of a particular bit x_n after observing a random bit in z can be approximated by:

$$f_1(x_n) = \begin{cases} +l_z & \text{with probability } \frac{1-p_z}{N} \\ 0 & \text{with probability } \frac{N-1}{N} \\ -l_z & \text{with probability } \frac{p_z}{N} \end{cases} \quad (5.8)$$

The large probability of $f_1(x_n)$ remaining unchanged is because we are very likely to observe a bit z_m that does not copy our particular x_n . After M bits in Z have been observed the distribution of the log-odds in x will be approximately Gaussian as it is the sum of many independent identically distributed terms:

$$l_x \sim \log \left(\frac{1 - p_z}{p_z} \right) \cdot \mathcal{N} \left(\mu = \frac{M(1 - 2p_z)}{N}, \sigma^2 = \frac{M}{N} \right). \quad (5.9)$$

5.2 Simple Error-Correcting Codes

(We have neglected factors close to one in the estimate of the variance.) The bit error rate is estimated by $\Pr(l_x > 0)$. Already at this stage we have enough to make a theoretical prediction for the probability of bit error of the parity code with $\mu = 1$ as a function of the rate of the code. This prediction is shown as the upper light grey line in figure 5.2(b), and the prediction is well matched by the empirical performance of bit copying (also in figure 5.2(b)).

Now we consider the decoding the parity of *two* variables. Given the log-odds of the values of x, y in l_x and l_y we would like to calculate the new evidence for l_x after observing a noisy evaluation of $x \otimes y = z$. We start by substituting:

$$p = \frac{1}{1 + e^{-l_z}}, \quad \bar{p} = \frac{e^{-z}}{1 + e^{-l_z}} \quad (5.10)$$

into the log-odds of equation 5.3 which simplifies to:

$$l_x = \log(e^{-l_z} + e^{-l_y}) - \log(1 + e^{-l_z - l_y}) \quad (5.11)$$

$$l_x \approx \max(-l_z, -l_y) - \max(0, -l_z - l_y). \quad (5.12)$$

This shows that the information is limited by the most uncertain bit in our inference which is sensible since any individual bit flip will change the parity. This is most easily seen by setting $l_x = 0$ (which corresponds to a completely uncertain bit) and seeing that $l_x = 0$ (we receive no new evidence) regardless of l_y . Striving for insight, rather than accuracy we assume that information will be transmitted, only if $|l_y| > |l_z|$. The average effect on the log-odds if it is already distributed $\sim \mathcal{N}(\mu, \sigma)$ can be described by:

$$f_2(x) = \begin{cases} +l_z & : \frac{2}{N}\Phi(v)(1-p) \\ -l_z & : \frac{2}{N}\Phi(v)p \\ 0 & : \text{otherwise} \end{cases} \quad (5.13)$$

where $v = \frac{l_z - \mu}{\sigma}$. The advantage of this stricter requirement is that we get to update our beliefs for two variables. The method immediately generalises to the parity of n variables:

$$f_n(x) = \begin{cases} +l_z & : \frac{n}{N}\Phi(v)^{n-1}(1-p) \\ -l_z & : \frac{n}{N}\Phi(v)^{n-1}p \\ 0 & : \text{otherwise} \end{cases} \quad (5.14)$$

5. ASSOCIATIVE MEMORY FROM ERROR-CORRECTING CODES

One can generate the connectivity for a parity code of length N by first generating the connectivity of a code of length $N - 1$ and then adding the last connection based on which connectivity will give the best mean improvement in the probability of bit error. The best connectivity for a code of length 1, is a simple bit copy.

In practice the approximations are quite rough and a correction factor $\zeta = 0.8$ had to be added to $v = \zeta^{\frac{l_z - \mu}{\sigma}}$ before good results were produced. Once the method begins to include the parity of half the bits in x then it is a dense encoder. Since the number of bits used in the parity scales with the block length of the code then the predicted probability of error should drop to zero with increasing block length, however codes constructed in this manner are still “bad” codes (using terminology from MacKay & Neal (1995)) whose rate also tends to zero with increasing block length (albeit very slowly).

We created connectivities which followed the above procedure and stopped when the procedure included the parity of half the bits in x . In figure 5.2(b) we show both the theoretical performance (lower light grey line) and the empirical performance (green line) of a parity code constructed this way. The empirical performance is markedly better than the predicted performance at rates above the transition to a dense encoding. This suggests that better modelling of the error probability might lead to improved results, but we leave this as future work.

5.2.4 Performance

In figure 5.2(a) we plot the results for a sigma-pi error correcting code on a binary symmetric channel. The input block length is $N = 500$, and the output length is in the range $M = 500 \dots 5000$. The connectivity ranges from “OR”ing 44(= b) different groups of 6(= a) bits “AND”ed together down to the much simpler connectivity $(a, b) = (2, 2)$. The figure clearly shows that for high rates a simple connectivity performs best, while low rates of communication have the best performance with high connectivity. Also note that the performance for some of the parameter settings is not shown, this is due to the lack of a single error in all the testing. We estimate that in these cases $p_x < 3 \times 10^{-7}$.

In figure 5.2(b) we show similar results for the parity code. At different rates the optimal connectivity varies, and we only plot results for the best mean connectivity at

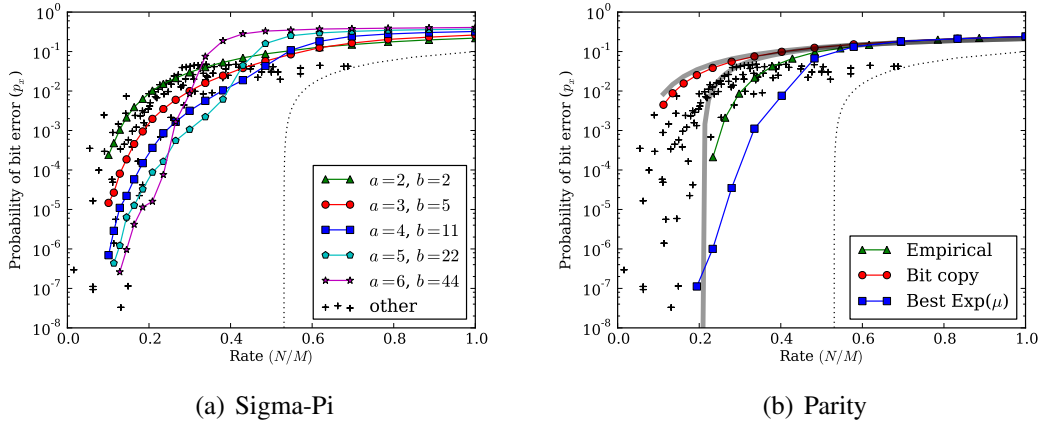


Figure 5.2: Performance of the neurally-inspired error correcting codes on the binary symmetric channel. Results shown are for a block length $N = 500$ and a probability of storage error $p_z = 0.1$. The Shannon limit is plotted in a dashed line, and compared against the performance of codes with different connectivity. The other codes plotted include RM codes, BCH codes and simple repetition codes. The grey lines for the parity code show the predicted performance for a bit-copying code, and the estimated optimal connectivity as estimated by the effect on the log-odds. The empirical performance for the optimal connectivity is also shown and is markedly better than predicted. We also plot the performance for connectivities sampled from an exponential distribution (using the best mean for each rate).

5. ASSOCIATIVE MEMORY FROM ERROR-CORRECTING CODES

each particular rate. This change in optimal connectivity is unfortunate for our stated goal of creating an error-correcting code that is able to function effectively at all rates as we allocate and de-allocate computational units to it. However on a practical level both codes have settings which appear near-optimal for rates in the range $\frac{N}{M} \in (\frac{1}{3}, \frac{1}{10})$. This still gives one sufficient flexibility to vary the number of computational subunits assigned to a particular error-correcting code by a factor of 3 and the probability of error improves by several orders of magnitude.

5.2.5 Efficiency

It is natural to find the most efficient use of the computational units when we use the concepts from error-correcting codes to create an associative memory. If we only care about the total information contained in the associative memory, then we can trade the number and reliability of the stored patterns given a fixed number of computational units. We need to trade off recalling more patterns with low fidelity against recalling fewer patterns with high fidelity. The information in a noisy binary vector is again given by equation 3.20:

$$s(p_x) = N (1 - H_2(p_x)) \quad (5.15)$$

and the efficiency (ρ) is the improvement in bits divided by the number of bits observed. In the error-correcting code there is no initial cue so $p_c = 0.5$:

$$\rho(p_z) = \frac{N (H_2(p_c) - H_2(p_x))}{M(1 - H_2(p_z))} \quad (5.16)$$

It is important to note that the efficiency is a function of the level of noise in the storage. In figure 5.3, we show the efficiencies of both the parity code and the sigma-pi code for a noise level of $p_z = 0.1$. We see that in spite of completely random connectivity we can still obtain an efficiency of 75% of the Shannon limit.

The striking thing is how unimportant a zero error rate is if we are only concerned about the efficiency of the code. For both the sigma-pi code and the parity code, the efficiency peaks at a rate of approximately 0.5. At this rate the probability of bit error is on the order of 0.01. The parity code is slightly more efficient and requires less computation. As a result of the better efficiency we will use the parity code to create an associative memory. The first step, which we cover in the next section, is to examine

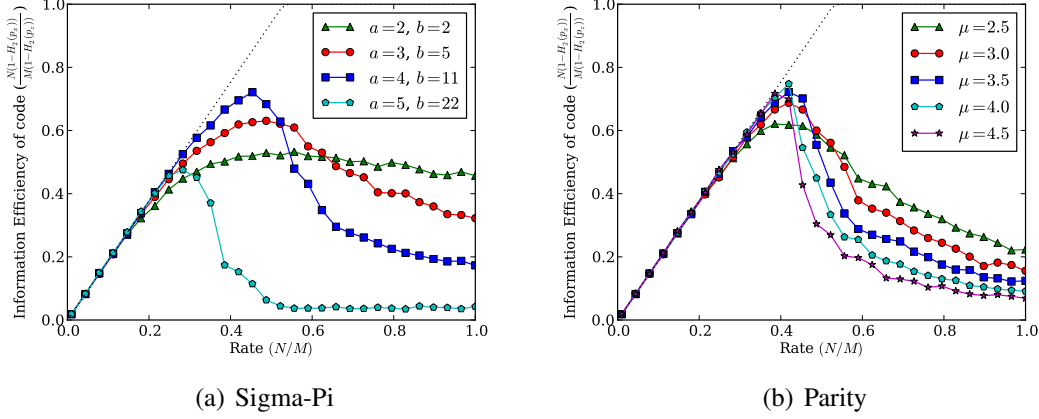


Figure 5.3: The information efficiency of the neurally-inspired error-correcting codes. These codes were created for a block-length $N = 500$ and probability of bit error $p_z = 0.1$. The dotted line shows the Shannon limit.

how the task changes when an initial cue is present. The extra information from the cue allows us to use less storage, and will form the basis for an efficient associative memory.

5.3 Cued error-correction

We now modify the task slightly so that the storage is now reliable ($p_z = 0$), and we have an initial cue for x . Both these modifications mean that we require less storage, and in particular we will focus on a storage that is smaller than the item we would like to store, with the extra information provided by the cue. We show the structure of the task in figure 5.4 (which can be contrasted with figure 5.1(d)).

For this section (and the next) we will only focus on error-correcting codes that are based on parity checks. In spite of the excellent probabilities of bit errors at low rates we will also not use the connectivities from the theoretically constructed distribution. Instead we will use connectivities sampled from an exponential distribution. These codes are efficient at relatively high rates (as shown in figure 5.3(b) and the exponential distribution gives a lower probability of bit-error at these high rates (as shown in figure 5.2(b)).

5. ASSOCIATIVE MEMORY FROM ERROR-CORRECTING CODES

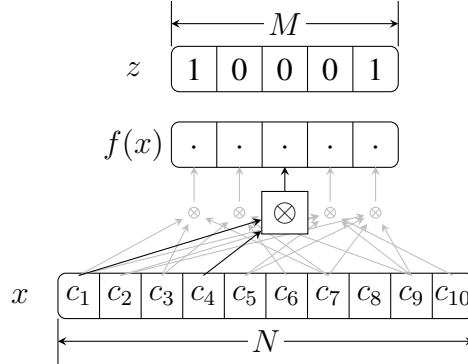


Figure 5.4: The cued error-correction task: we are given an initial noisy cue c , the noiseless results of some random parity functions in z and must recover the original x .

5.3.1 Minimum connectivity

Given that the new task has an initial cue, we no longer have to store the parity of single bits but can jump directly to the parity of several bits. The minimum number of bits that we take the parity of will depend on the strength of the cue. For very strong cues, we need to take the parity of many bits. It is important that we find the correct minimum; if we initially try to store the parity of too few bits, then we need much more storage to infer the original pattern, while using the parity of too many bits makes inference difficult.

In this section we will show what the minimum number of bits should be for a cue of a given strength. Instead of an exact derivation we use our knowledge of several of the properties that a solution must have to bound the required function. We then show that the simplest function which satisfies all the requirements is a good approximation to empirical results.

If we let n denote the minimum connectivity, then we seek a function of the given cue strength which returns the required minimum connectivity: $n = f(p_c)$. We know that this scaling relationship should be symmetrical for cues with a very high probability of bit-flip and a very low probability of bit flip:

$$f(p_c) = f(1 - p_c). \quad (5.17)$$

We also know that for a completely uncertain cue, we need to use the parity of single

bits:

$$f(0.5) = 1. \quad (5.18)$$

Finally, as the cues tend to be more reliable the expected number of incorrect bits that a parity check is connected to should tend to a constant. This constant should be not zero, for then nothing is learnt, nor should it tend to an average of more than one incorrect bit per parity check for then it is impossible to identify the incorrect bits:

$$0 < \lim_{p_c \rightarrow 0} np_c < 1. \quad (5.19)$$

The simplest equation that satisfies all of the three preceding equations is:

$$n = \frac{1}{4p_c(1 - p_c)}, \quad (5.20)$$

but this is not a unique solution; for example the following also satisfies the requirements:

$$n = \frac{\frac{p_c}{1-p_c} + \frac{1-p_c}{p_c}}{2}. \quad (5.21)$$

This solution (and all others) will only differ from equation 5.20 by a constant factor for very small or very large levels of bit-flips in the cue ($p_c \approx 0$ or $p_c \approx 1$). For our purposes identifying the scaling up to a constant factor is sufficient.

In figure 5.5 we show that equation 5.20 accurately predicts the required minimum connectivity. We created increasingly reliable cues for increasingly large patterns in such a way that the number of expected incorrect bits remained constant (here $Np_c = 10$). We then allocated enough storage so that we could be sure the task was solvable:

$$M = 2.5NH_2(p_c), \quad (5.22)$$

even though our previous results suggest that the cued error-correction task should be solvable at an efficiency of 70%, we were cautious and chose a lower efficiency of 40% to allow a larger (and easier to find) region of success. We then tested several different levels of minimum connectivity. At each level of minimum connectivity we decoded several cues and if the decoding was successful for more than half of the cues then we deemed the level of connectivity to be correctly set. It is worth highlighting how efficient this scheme is for large patterns with very low levels of noise. The largest patterns we considered had $N = 10^6$, yet we only needed $M = 451$ bits to

5. ASSOCIATIVE MEMORY FROM ERROR-CORRECTING CODES

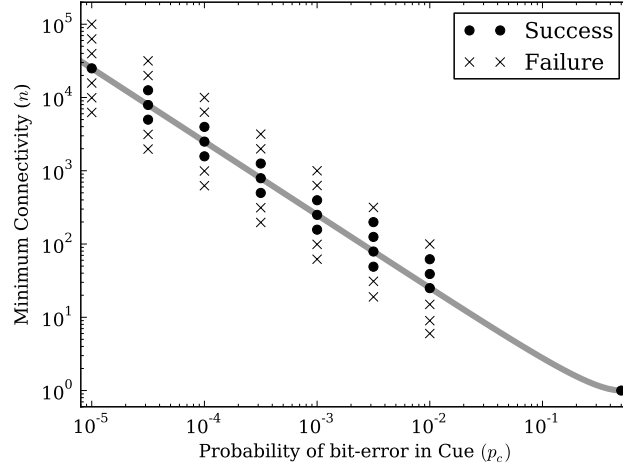


Figure 5.5: The required minimum connectivity for increasingly reliable cues. We also scaled the pattern size so that the average number of incorrect bits in a cue remained constant – $Np_c = 10$. Equation 5.16 which satisfies all the known constraints of the minimum connectivity (but not uniquely) is plotted in grey and matches the empirical results.

successfully correct the (typically) 10 incorrect bits. Here the probability of bit error was $p_c = 10^{-5}$, and the ideal minimum connectivity is 25000 bits (beyond these limits we ran into problems with insufficient floating point accuracy).

5.3.2 Results

In figure 5.6 we show the error rates for the cued error-correction task when $p_c = 0.1$ and a pattern size $N = 1000$. We consider cleaning up the noisy cue for storage sizes in the range $M = 500 \dots 2000$, with different levels of mean connectivity. While the error rate drops monotonically as the rate decreases, we note that there is a natural maximum for the efficiency of the cued error-correction task. Here the maximum is still roughly 70% efficient and occurs for a rate $N/M \approx 1.5$ code. If we add more storage than is optimally efficient then the error rate will still fall rapidly but recall is already quite reliable and the further reduction in error does not translate into many bits of new information. If we use less storage than is optimally efficient then the statistical evidence is inconclusive and recall is unreliable. The exact position of this maximum

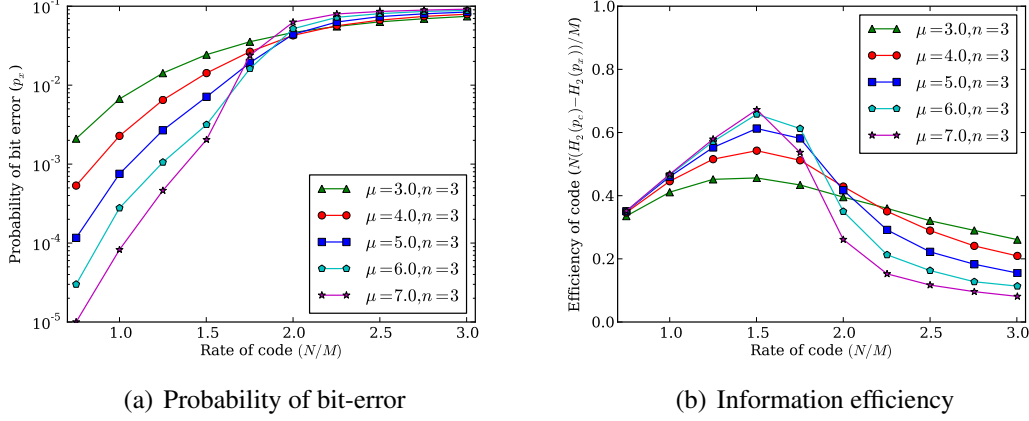


Figure 5.6: The change in performance as more storage is added to the cued error-correction task. For these figures $p_c = 0.1$, the input code size $N = 1000$ and we did not include parity checks of fewer than 3 bits. We then varied the amount of available storage M and decoded many input vectors to get (a) the empirical probability of bit error p_f , and also (b) the information efficiency.

will depend on the reliability of the storage (here we have used reliable storage, $p_z = 0$) and the strength of the cue (here $p_c = 0.1$). It is also likely that the efficiency could be improved with a better connectivity distribution.

If we are to use the cued error-correction hardware to create an associative memory, and the only thing that matters is to make efficient use of the hardware then there is a natural amount of cued error-correction hardware to allocate per memory stored. In a practical application there might be other considerations, such as a required probability of bit error in the final recall, or a minimum number of patterns to be stored in a fixed amount of hardware. In the next section we will look at creating an associative memory which uses cued error-correction hardware and examine the trade-offs in the design of such a memory.

5.4 Population codes

In this section we create an associative memory by storing many cued error-correction tasks. When an item is stored in the memory we will allocate new cued error-correction

5. ASSOCIATIVE MEMORY FROM ERROR-CORRECTING CODES

hardware for the item. We also allocate hardware which stores a portion of the pattern to indicate when the pattern should be recalled. The simple copy which best matches the initial cue then activates its associated cued error-correction to clean up the cue.

In an error-correcting code the same hardware is repeatedly used to encode and decode many different messages. When using our simple error-correcting codes to create an associative memory we will group a single pattern together with a single error-correcting code. This is in keeping with the neurally-inspired design goals where many simple pieces of hardware, each of which has the ability to memorise a single bit, can act together to perform complicated calculations. If we wish to add or remove a pattern from the associative memory we will talk as if we could physically add or remove the computational units. It is likely that we would not have such detailed physical control of the system so achieving this in practice would require extra information which would keep track of the populations of computational units that have not been allocated. However this extra book-keeping is not the focus of this chapter and we will leave such considerations for future work.

We have called this associative memory a population code to emphasize the two most salient points about it. Each pattern is stored by activating a single population of computational units and at the heart of each population there is an error-correcting code. An overview of the population code is given in figure 5.7.

For the r^{th} pattern we denote the small simple copy of the pattern by z_{rs} and the associated error-correction hardware by z_{re} . As was shown in the previous section, the cued error-correction is not perfectly efficient and there is some redundant information in the representation. The redundancy means that one could potentially select the pattern by calculating $\Pr(z_{re} | c)$. Equation 5.20 shows that the mean number of incorrect bits per parity check for low-noise cues is approximately $1/4$. Modelling the actual number of incorrect bits per parity check with a Poisson distribution, we see that the probability of no incorrect bits in a parity check is $e^{-1/4} \approx 0.78$. For most realistic tasks this represents enough redundant information in the cued error-correction storage to make the small copy of the pattern unnecessary. We find the (slightly artificial) split illuminating, as the required simple copying hardware scales with the number of patterns stored, while the required size of the cued error-correction scales with the uncertainty in the cue. Keeping these two things separate allows for the different scaling of the pattern size, the cue uncertainty and the number of patterns stored, while still

maintaining predictable performance. If the redundant information was used to select the pattern and a better solution to the cued error-correction task was found then there would be less redundant information with which to select the correct pattern and the probability of bit error might actually increase.

5.4.1 Allocation

We need to decide how to allocate resources between the simple units and the complex units. We will use m_s to denote the number of simple units and m_e to denote the number of error-correcting units for an individual population. Note that in this section we do not consider different costs for simple or complex hardware, each is only a single bit. After storing R patterns we will then have a storage size of:

$$M = R \cdot (m_s + m_e). \quad (5.23)$$

We first focus on estimating the probability of failure from choosing the wrong pattern to recall. To simplify the analysis we assume that when examining the simple copying hardware the correct pattern will differ from the initial cue in exactly $p_c m_s$ bits (which is the expected mean). Since the other patterns are uncorrelated random binary vectors we can use Chernoff's inequality (Mitzenmacher & Upfal, 2005) to bound the probability that another pattern will have fewer errors:

$$\Pr(\text{Binomial}(n, p) < k) \leq \exp\left(-\frac{1}{2p} \frac{(np - k)^2}{n}\right). \quad (5.24)$$

The other patterns are uncorrelated which implies $p = 0.5$, and we wish to estimate the probability that a single uncorrelated pattern has fewer than $p_c m_s$ errors when predicting the cue. This gives the probability of failure for a single uncorrelated pattern:

$$\Pr(\text{failure} \mid 1 \text{ other pattern}) \leq e^{-m_s(\frac{1}{2} - p_c)^2} \quad (5.25)$$

For a very small probability of failure for a single pattern we can estimate the probability of failure when there are R patterns by multiplying by R (since it is improbable that two failures will occur in a single trial):

$$\Pr(\text{failure} \mid R \text{ other patterns}) \leq R \cdot e^{-m_s(\frac{1}{2} - p_c)^2} \quad (5.26)$$

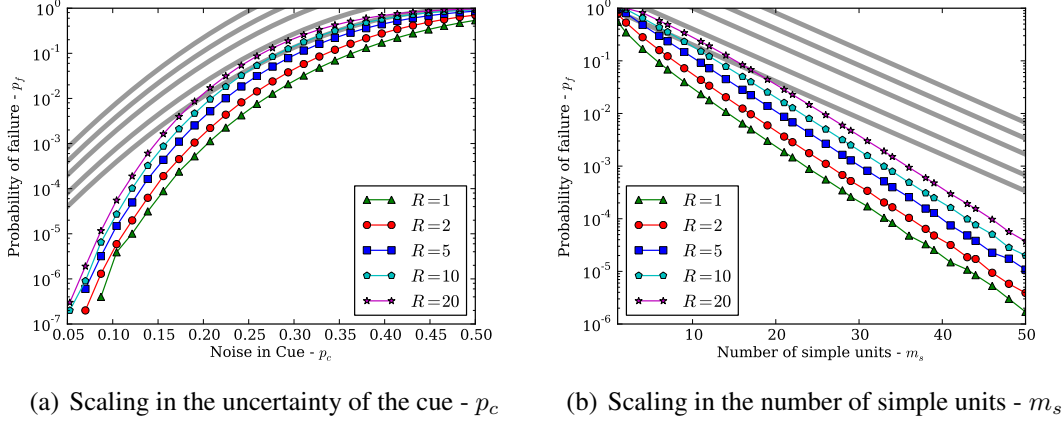


Figure 5.8: The empirical error of identifying the correct pattern out of several other random binary vectors given only a small noisy copy of the pattern. We plot the results for several different numbers of patterns R , and show the corresponding upper bounds from equation 5.27 in light grey lines. For (a) the number of units is fixed at $m_s = 50$, and (b) has the cue noise fixed at $p_c = 0.1$.

Denoting this probability by p_f we can obtain a simple scaling relationship:

$$m_s \geq \frac{\log\left(\frac{R}{p_f}\right)}{\left(\frac{1}{2} - p_c\right)^2}. \quad (5.27)$$

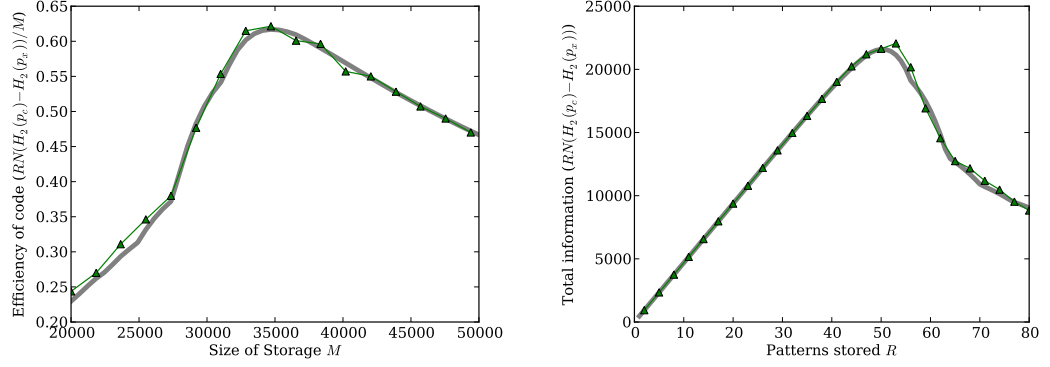
This shows that the error rate falls off exponentially as the number of simple units is increased. Note that this rate of exponential decrease might still be slow if the cue is very noisy (i.e. $p_c = 0.5 - \epsilon$). We show the results of the approximation in figure 5.8 and note that it is a relatively conservative upper bound (particularly for low levels of cue noise).

We can marginalise out over the two possible conditions to find the final probability of bit error. We let \bar{A} represent the activation of the incorrect population, and A represents the activation of the correct population:

$$\Pr(\text{Bit error}) = \Pr(\text{Bit error} | \bar{A}) \Pr(\bar{A}) + \Pr(\text{Bit error} | A) \Pr(A) \quad (5.28)$$

Even if the wrong error-correcting code is activated (\bar{A}) however, only Np_c bits will be flipped on average. This means that the probability of bit error in this condition is

5. ASSOCIATIVE MEMORY FROM ERROR-CORRECTING CODES



(a) Efficiency of the code as the storage size M is varied while the number of patterns is fixed $R = 50$. (b) Total information recalled by the network while the size of the storage is fixed $M = 35000$ but the number of patterns can vary.

Figure 5.9: Patterns of $N = 1000$ bits were stored and recalled using cues that have 10% noise. We investigated the performance for different storage sizes M , and different numbers of patterns R . The optimal network stored 50 patterns and used 35000 bits of storage. For both figures the grey line represents the predicted information recalled using the empirical error rate from figure 5.6(a) and equation 5.26, while the green line represents the actual information recalled.

$2p_c(1-p_c)$ (since some bits are already incorrect in the cue and we assume that the new flipped bits are independent of the correct pattern). If the correct error-correcting code is activated (A) then there will still be bit errors from the error correction hardware. If we denote the probability of recalling the wrong pattern by p_s and the probability of bit error in the cued error-correction by p_e , then the final probability of bit error in the recall is:

$$p_x = 2p_c(1-p_c)p_s + p_e(1-p_s) \quad (5.29)$$

We can use the results from figure 5.6(a) to estimate the probability of bit error from the error-correcting code p_e when using the correct population to clean up the cue and use equation 5.26 to estimate the probability p_s of choosing the incorrect pattern to decode.

If the computational units are expensive then we do not want to maximize the total information recalled, but rather the information recalled per computational unit. In

figure 5.9(a) we created a population code to store $R = 50$ patterns and recall them with a cue containing 10% noise ($p_c = 0.1$). From theory behind the binary symmetric channel we know that the cue contains approximately 531 bits of information and we require at least another 469 bits from our network to complete the pattern. We then explored the recall efficiency for different amounts of storage, ranging from 400 bits per pattern to 1000 bits per pattern. The efficiency is low for small amounts of storage, but rapidly increases as the storage transitions to reliable operation. Once the network is operating reliably, allocating more storage decreases the efficiency.

We can also consider fixing the total number of computational units, and optimising the number of patterns to store. The results are shown in figure 5.9(b). Again we chose $N = 1000$ bit patterns to store and recalled them with cues containing 10% noise. We fixed the total number of units at $M = 35000$, letting the total number of patterns stored range from $R = 1$ to $R = 80$. The maximum information recalled occurs when $R = 50$. If we store fewer patterns then, although they can be recalled with high fidelity, there isn't enough information in the small number of patterns. Alternatively if we store more patterns then there isn't enough hardware per pattern to ensure successful recall.

We see that figures 5.9(a) and 5.9(b) are consistent and both have a maximum for the situation where $R = 50$ patterns are stored in a storage of $M = 35000$. Examining the situation in more detail we find that, out of the 700 bits per pattern, it is optimal to allocate $m_s = 62$ bits for the simple hardware which detects the correct pattern to activate. Using 638 bits to correct the $N = 1000$ bits of information gives a rate of approximately 1.5, which was shown to be the optimal rate in figure 5.6(b). This is a fairly general observation; for the typical parameters that one would use this associative memory for (large patterns with low-noise cues) the overwhelming majority of the storage will be allocated to the error-correcting codes. If we want efficient use of the hardware, then it is important to use the error-correcting codes at their most efficient rate.

The storage required that is required to identify the correct error-correcting codes grows logarithmically in the number of patterns R , yet forms a small percentage of the overall storage. While the scaling is not truly linear we can use our physicist's sense of approximation and declare the efficiency to be **0.63** since this is the efficiency of a typical code which allocates 10% of its units to simple copying (as in figure 5.9(a)).

5. ASSOCIATIVE MEMORY FROM ERROR-CORRECTING CODES

Other settings will change the ideal ratio of low-connectivity units to high-connectivity units which will affect the efficiency. For settings where the efficiency is low it will be better to choose other architectures.

We should note that it is possible our other networks also have a similar scaling, which is theoretically *not* linear. Even for relatively small patterns ($N \approx 100$), this is difficult to differentiate from true linear scaling, which would discard the $\log R$ term. In practice this is a non-issue as an astronomical number of patterns would have to be stored before $N \not\approx N + \log R$.

5.5 Conclusion

This chapter used concepts from error correcting codes to create an associative memory that was particularly efficient for low-noise cues. We first examined simple error-correcting codes which are a form of low-density generator matrix codes. These are simple to encode and decode, and are robust in the sense that the loss of a small number of units will not catastrophically affect the error rate. Unfortunately these codes are far from the Shannon limit. However if we care only about the efficient use of the hardware, then these simple codes perform well. Using hardware with an intrinsic noise of 10% we can achieve an efficiency of 75% of the Shannon limit just by driving the final error down to 0.5%. Thus our focus on building an efficient associative memory is very different from standard error-correcting codes as we can tolerate (relatively) high final error rates. From our perspective driving down the error rate further is sub-optimal as we can make more efficient use of our hardware by storing more patterns in our associative memory.

Associative memory needs an initial cue to select the correct pattern, so we briefly looked at an error-correcting code that bootstrapped from an initial cue. We called this the cued error-correction task (although it can also be thought of as a standard systematic code with different noise levels in the pattern and the parity bits). The initial cue reduced the required number of bits to be stored.

We then constructed an associative memory out of many cued error-correction tasks. We also stored a copy of a portion of the pattern so that we would know when to recall a particular pattern. The memory is inelegant as it needs to add new hardware for each new pattern stored. The memory also requires two stages to perform recall, the

first round is winner-take-all to find the pattern which best matches the initial cue. The second round then uses only the stored error-correcting code of the winning pattern to clean up the cue. The big advantage is that if we know how informative the cues will be in advance then we can use fewer than N bits to store the error-correcting code of an N -bit pattern (with the remainder of the information provided by the cue). A large amount of the increase in efficiency is also due to the lack of interference between different patterns.

The associative memory given in this section, will be relatively inefficient for small patterns and weak cues ($p_c = 0.5 - \epsilon$), since the required number of simple units grows $O\left(\frac{1}{\epsilon^2}\right)$. Conversely this associative memory performs extremely well with large patterns which are recalled with strong cues since the storage required is a tiny fraction of the pattern stored. We also believe that the performance could be further improved with a better distribution of connectivity (other than the exponential distribution).

While the memory has several stages to recall, it is still robust to the hardware failure of any individual unit. The memory also has low complexity as its description length is very short. To illustrate this, consider that the exact connectivity of the network does not need to be described to successfully construct such a network. One only need sample random connectivity and the network will work with very high probability. All that one needs to describe is the distribution of connectivity (in this case we used an exponential distribution with a mean μ and minimum connectivity n), the input size N , the storage size M , the split between simple units or cued error-correction units and finally, the encoding function $f(x)$. With this short description one can construct an associative memory that requires an amount of storage which is within a factor of two of the theoretically-optimal associative memory.

The increase in efficiency from 0.36 in chapter 3 to 0.63 here can largely be attributed to the lack of pattern interference. This increase in efficiency could provide an extra reason for the localization of information in neocortex (Swindale, 1990) (in addition to the minimization of wiring length (Chklovskii *et al.*, 2002)).

We now look at an associative memory which performs in continuous time and has the exciting property that it can recall many patterns simultaneously.

5. ASSOCIATIVE MEMORY FROM ERROR-CORRECTING CODES

CHAPTER 6

Associative Memory in Continuous Time

In this chapter we examine the associative memory task in the continuous time setting¹. We again store random patterns, but now instead of random binary vectors these are patterns sampled from a Poisson process. These patterns are similar in form to neural spike data and, as such, we will focus on a method of recall that is biologically plausible and use neural terminology freely even though there are several aspects of this work which need refinement before it is completely biologically plausible. In figure 6.1 we show an overview of the required task. In figure 6.1(a) there are some example patterns that we are required to store. Each pattern is coloured a different colour for ease of identification, but uses the same set of neurons. We chose periodic patterns to allow the network sufficient time to recall the entire pattern (the pattern can take several periods before it is recalled in its entirety). The periodicity of each pattern is allowed to vary, so that different patterns could be recalled at different rates. We also impose the weak constraint the each pattern has the same number of spikes (regardless of the period).

We give the network an initial cue consisting of a noisy pattern with spikes missing and jittered in figure 6.1(b). If the network completes the pattern then it has done useful computation. The useful computation can be measured in bits by considering the difference in information between the initial cue and the final recall. The mutual information between the final recall and the ideal pattern represents how much information the network can reproduce, while the mutual information between the cue and the ideal pattern represents how much information the cue contains. The difference between these two quantities is then how much information the network has added.

¹The task and solution for this chapter build on the Concurrent Recall Network contained in Wills (2004)

6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME

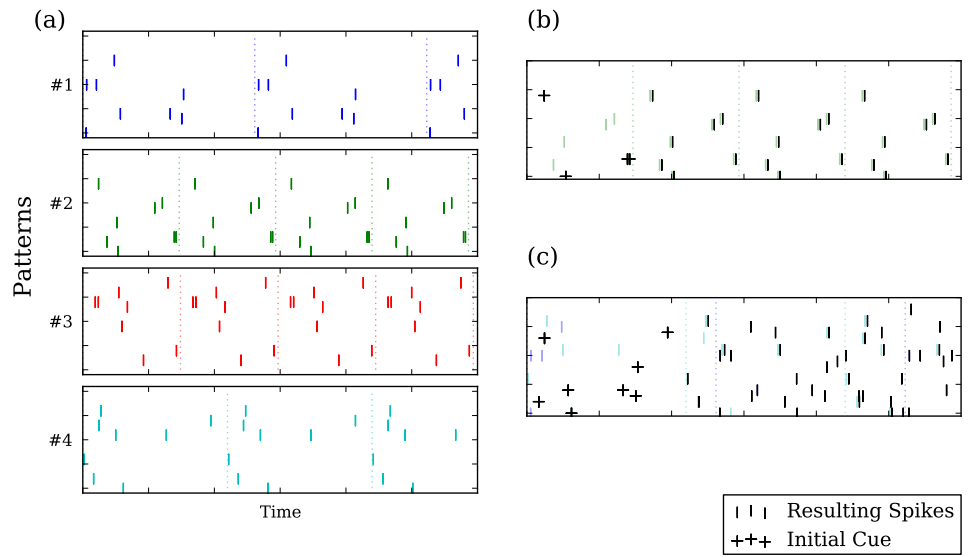


Figure 6.1: Overview of the task: We would like to store several periodic patterns in a network and give examples in (a). Each pattern can have a different period, indicated by the dotted lines, but we assume has the same number of spikes. (b) Given a cue in the first period consisting of a small number of spikes from a particular pattern we would like the network to fill in the rest of the pattern in succeeding periods. (c) If cues from several patterns are present we would like all the relevant patterns to be recalled simultaneously.

This is very similar to the measure used in previous chapters where the increase in information is the important quantity.

An ideal solution to this task would also be able to recall several patterns simultaneously, given only a small portion of each active pattern. We illustrate this possibility in figure 6.1(c). Using simple superposition we overlay the cues from two patterns and the network can complete both patterns simultaneously. Note how the ability to recall more than one pattern has increased the number of states the network can be in. If four patterns are stored and only a single pattern can be recalled, then the network can be in only four states. However if the network can recall two patterns simultaneously then the network can be in $\binom{4}{2} = 6$ states.

In our proposed architecture which solves this task, we use neurons which are composed of many individual dendrites. Each dendrite is wired to detect only a single pattern (although it would be useful future work to determine how best to get a single dendrite to detect several patterns simultaneously). If a dendrite detects that its pattern is present then it will immediately signal the neuron to fire. If we want to add a pattern to the network then we will add a dendrite to each neuron which participates in the pattern. While explicitly adding dendrites to a neural network is biologically implausible, we have followed this approach for its conceptual simplicity. One can imagine other ways of achieving this without explicitly adding dendrites, for example having dendrites whose synapses have very low weights, which then get activated in a simple Hebbian manner when a new pattern is stored (Bliss & Lømo, 1973).

For a dendrite to detect that its pattern is present we make use of delay lines (Gerstner *et al.*, 1993; Hopfield, 1995; Wills, 2004). This is similar to Polychronization (Izhikevich, 2006; Vertes & Duke, 2010), Synfire chains (Aviel *et al.*, 2002) and the more sophisticated Synfire braids (Bienenstock, 1995). Our contribution is to quantify how many bits such methods can convey and closely examine the trade-offs involved in setting various parameters. Each delay line is connected to a neuron that participates in the pattern with an appropriate delay such that if the pattern is present then all preceding postsynaptic potentials will arrive at the same time (this is illustrated in figure 6.2(b)). We then use a model of a dendrite as a coincidence detector.

Our dendritic model is inherently noisy and the neurons will generate spikes in the absence of any input. Moreover even when given sufficient input to spike reliably there will be variation in the precise spike time. This inherent noise is important when

6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME

measuring information contained in continuous times. If there was no noise then a network could produce an arbitrarily large amount of information in the relative spike times of just two precisely-timed spikes. It also forces us to consider various strategies to cope with this noise.

Loosely speaking the dendritic model can be thought of as sampling from a posterior belief in the presence of its pattern, although we have chosen a simpler (but incorrect) method of integrating the evidence over time. Each time a postsynaptic potential is received this can be viewed as evidence that the pattern is present. If this evidence is in log-odds form then it behaves linearly and can be added to the current evidence. This evidence will then increase the posterior probability in an exponential (or close to exponential - see equation 6.7) manner and will motivate our choice of the LNP model from (Simoncelli *et al.*, 2004) (LNP neurons are so named for the three stages of the model; the Linear stage, the Non-linear stage and a Poisson process).

This chapter will cover the many trade-offs when using such a network for the associative memory task. A major trade-off is the number of patterns one stores and the number of patterns that are recalled. If we denote the number of patterns stored by M and the number of patterns recalled by m then the number of states the network can be in is $\binom{M}{m}$ (assuming there are no failures of the network which we will discuss later). We will show that as the number of possible states increases it negatively affects the accuracy of individual spike times. For different tasks it might be more natural to either encode information in spike times or in the set of active patterns, and we will examine the different settings which maximise either measure of information. The number of spikes in a pattern also has a large influence on the performance of the network; too few and the pattern is liable to die out, while too many reduces the number of patterns that can be simultaneously recalled.

Finally we examine the form of the cue which results in maximum information recalled. Our performance measure rewards accurate recall, but penalises precise cues, so this amounts to finding the noisiest cue from which successful recall is still possible.

6.1 Related work

Before presenting the model in detail we first examine related work that contributed to the development of the model. Synfire networks (Abeles, 1991) were one of the first

network models which allowed distributed representations of many patterns simultaneously. As such they are covered first in this section. Polychronization (Izhikevich, 2006) is a more sophisticated variant of this network as it uses a more sophisticated neural model and allows the delay lines between neurons to vary. Finally the Concurrent Recall network (Wills, 2004) is covered which bears the most resemblance to the work presented here. While the concurrent recall network uses a very simple neural model, this chapter is novel as it uses a more sophisticated model, better suited to measuring spiking accuracy. This chapter is also novel in quantifying possible recall capacity using information theoretic measures.

6.1.1 Synfire networks

A synfire chain is a network of integrate and fire neurons which has been wired to create a number of different pools of neurons. Each neuron may participate in many different pools and each pool represents a state of information. These pools are linked together so that coherent activity in one pool will lead to the activity in the connected pool (and the activity in the preceding pool will subside). The synfire network operates in a synchronous fashion as it is assumed that transmission delays between any pair of neurons is identical, although this is not essential (Bienenstock, 1995). This leads to a burst of spikes travelling to the next pool and creating a similar burst of spikes. The synfire chain requires inhibitory neurons to prevent the oscillatory behaviour from proliferating.

If one embeds many synfire chains in a network then it is crucial to get the size of the pools correct. If the pools are too small then they will not be robust to noise, while pools that are too large will reduce the capacity of the network. It is also important to tune the average connectivity within pools to ensure maximum performance (Herrmann *et al.*, 1995).

There is some biological evidence in favour of synfire chains (Ayzenshtat *et al.*, 2010; Ikegaya *et al.*, 2004), although this is controversial (Oram *et al.*, 1999). The evidence needs to be carefully interpreted as many possible models can lead to patterns of precise reproducible spike patterns, not just synfire chains.

6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME

6.1.2 Polychronization

Synfire chains use constant time delays. Izhikevich's polychronization (Izhikevich, 2006) generalises the concept by allowing different delays between neurons. The neural model favoured by Izhikevich is more complex than an integrate-and-fire model, but is still comparatively simple (Izhikevich, 2004) while allowing a wide variety of behaviours. It is a canonical model of the Hodgkin-Huxley model (Hoppensteadt & Izhikevich, 2003), and is described by the following differential equation:

$$\begin{aligned}v' &= 0.04v^2 + 5v + 140 - u + I \\u' &= a(bv - u).\end{aligned}\tag{6.1}$$

The following rule is applied to determine whether the neural threshold has been exceeded and a spike should be generated:

$$\text{if } v \geq 30\text{mV, then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d. \end{cases}\tag{6.2}$$

The model is simple as it depends on only four parameters (a, b, c, d) and its ease of simulation allows large-scale simulations (Izhikevich & Edelman, 2008). An unattractive feature of this model is that it is quiescent and in the absence of external input the neurons will remain quiet.

Izhikevich (2006) creates a simulation of 1000 neurons with random connections between them. The delays are sampled uniformly from the range 1ms to 20ms. The model also includes inhibitory and excitatory neurons in biologically realistic ratios. After 24 hours of simulated time the network has settled into a state with many groups of spatio-temporal patterns. Neurons can participate in multiple groups and it appears that the network successfully learns these representations. A criticism of the work is that only 2 spikes are required to trigger a spike which is an exceedingly low threshold.

Vertes & Duke (2010) provides evidence that the storage capacity of these networks can be greatly increased through network topologies other than purely random (for example a randomly chosen small-world topology has roughly an order of magnitude greater capacity). This provides another direction for interesting future work; to discover the network topology which optimises storage capacity.

Izhikevich repeatedly stresses that a network with delays is infinite-dimensional. As a trivial example he states that the system described by the delay equation $x' =$

$-x(t-1)$ needs to have the initial conditions specified over the entire interval $[-1..0)$. While technically correct this is misleading since there is clearly noise in any physical neural system. If a neural system is performing any sensible notion of computation then this computation should be robust to the typical level of noise in the network.

6.1.3 Concurrent Recall Network

The concurrent recall network (Wills, 2004) uses a simple neural model which can be thought of as an extension of the sigma-pi function into continuous time. The model is very similar to the one presented here. For each pattern to be recognised a dendrite is allocated. The dendrite functions as a coincidence detector and generates a spike if at least s spikes have been received in the preceding several milliseconds.

The delays inherent in the network are also allowed to vary in a continuous fashion, but with a fixed level of noise. Wills (2004) reports an unavoidable phase precession during recall, although there are several means to avoid this, e.g. lengthening the delay lines to eliminate the bias or requiring a higher threshold before the dendrite spikes.

Wills (2004) analyses the capacity of the Concurrent Recall Network (CRN) as a function of network size and the relationship between patterns stored versus recalled. The empirical results are supported strongly with straightforward theoretical derivations and inspire much of this chapter.

This chapter will extend the work of Wills (2004) by using a more sophisticated neural model and carefully analysing the possible information that can be extracted from observing the network's spikes. Our results should be considered a lower bound however as it is possible that more information can be encoded through the clever embedding of further structure in the network. One such example is provided by Wilson (2009) who embeds many Hopfield networks within a single CRN.

6.1.4 Self-organising recurrent networks

This chapter does not cover any learning. All the patterns are known exactly beforehand and can be inserted into the network through the allocation of more dendrites. This is perfectly acceptable in terms of understanding network capacity, but it is unsatisfying given that biological networks must learn these representations.

6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME

Spike-time dependent plasticity (STDP) is a plausible learning mechanism (Markram *et al.*, 1997) with lots of experimental evidence. Rolston *et al.* (2007) cultured 50000 dissociated neurons together which then had evidence of temporally-precise spiking patterns (the neurons lacked the layered cortical structure). These results are suggestive of accurate spike times being an emergent feature of networks with STDP. (Lubenov & Siapas, 2008) also provides evidence for this: simulations of networks with different random delays are well-behaved in that populations that are over-synchronised (epileptic) end up being less synchronised, while networks that are randomly connected tend to synchronise.

The self-organising recurrent network (Lazar *et al.*, 2009) is interesting as it suggests that STDP is insufficient by itself to create a regime in which learning is easily achieved. Instead additional homeostatic mechanisms are also required (Watt & Desai, 2010) such as synaptic scaling and intrinsic plasticity. Synaptic scaling rescales all the synaptic strengths so that their total sum is kept constant. Intrinsic plasticity refers to a dynamic process whereby a neuron that has not spiked recently gradually becomes more likely to spike, even in the absence of any input. These different homeostatic mechanisms suggest interesting extensions of the current chapter.

6.2 Description of the model

For our purposes, we define a neuron as a collection of dendrites, each of which generates spikes individually (see figure 6.2(c)). Each dendrite generates spikes according to a linear-nonlinear-Poisson (LNP) model (Simoncelli *et al.*, 2004). When a single dendrite generates a spike, the whole neuron spikes and all the dendrites enter a refractory period. Other than this collective reset behaviour all dendrites are considered separately. An alternative (but mathematically equivalent) view is that each dendrite outputs a firing rate which is then summed over all dendrites by the neuron, the neuron then spikes according to an inhomogeneous Poisson process with the summed rate.

There is evidence that dendrites can be functionally separate (Branco *et al.*, 2010; Golding *et al.*, 2002; König *et al.*, 1996; Liu, 2004; London & Häusser, 2005; Polsky *et al.*, 2004; Softky, 1994) but our choice of functionally-separate dendrites is primarily to aid the easy insertion of a pattern into the network.

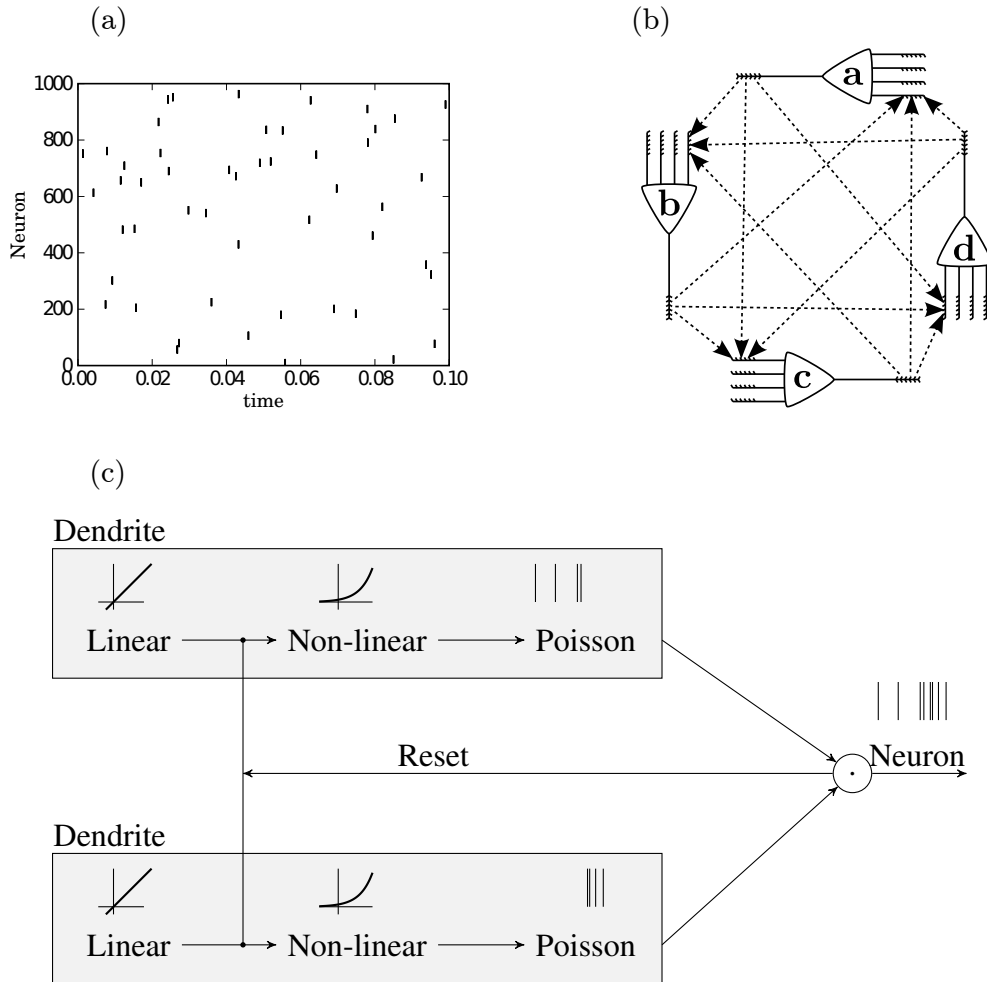


Figure 6.2: Overview of the patterns, how they are stored, and the neural model used in this chapter. In (a) we show a typical pattern; it is a sparse code as only 50 neurons fire in a 100 millisecond period out of a population of 1000 neurons. In (b) we show a schematic of the wiring of a pattern of firing ABCD. Each neuron has a single dendrite dedicated to the detection of the pattern. Once the pattern is started it will repeat indefinitely. The transmission delays must be set up so that spikes from the three preceding neurons will arrive simultaneously at the fourth. (c) The structure of the neural model. Each dendrite has an LNP model, and the neuron spikes whenever an individual dendrite spikes. All dendrites on a neuron are reset whenever the neuron spikes. After Simoncelli *et al.* (2004).

6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME

We define ψ_n to be the set of all spike times of neuron n :

$$\psi_n = \{\tau | \text{neuron } n \text{ spiked at time } \tau\}. \quad (6.3)$$

If ϕ represents the time that a signal takes to travel from the presynaptic neuron to the postsynaptic dendrite, then we can define the spike sequence arriving at a dendrite from a particular neuron as β :

$$\beta(n, \phi, t) = \sum_{k \in \psi_n} \delta(k + \phi - t). \quad (6.4)$$

This sequence of arrivals is modelled as a sum of Dirac delta functions, with each function positioned on a spike time plus the delay ϕ . We define ϕ_{ij} to be the delay from the j^{th} presynaptic neuron to the i^{th} dendrite (and to simplify the notation we will omit the index specifying which postsynaptic neuron it is). If there is no delay line from the j^{th} neuron to the i^{th} dendrite then we set $\phi_{ij} = \infty$. The voltage of the i th dendrite changes according to:

$$\frac{dv_i}{dt} = \frac{1}{\lambda} (v_{\text{EQ}} - v_i(t)) + \sum_j w_{ij} \beta(j, \phi_{ij}, t), \quad (6.5)$$

where w_{ij} is the synaptic weight. Equation 6.5 therefore represents simple exponential decay to an equilibrium voltage with a time constant of λ , and whenever a postsynaptic potential is received the voltage is increased by w_{ij} . Here we use a single constant for all the synaptic weights and denote this by w . We anticipate that there could be versions of the model with variable weights; where a synapse is stronger if it gives more reliable evidence that the pattern is present.

A central idea of this model is that each individual spike can convey considerable information through its timing. The more precisely timed a spike is, the more information content it can convey. As an extreme example, if we imagine that all spikes are generated, transmitted and detected with microsecond precision then a neuron firing once per 20 ms could convey 14 bits per spike. We include in the model a parameter ψ which determines the temporal precision of the delay lines. We assume that the exact delay for each spike is noisy and has a standard deviation proportional to the mean delay time. For this chapter we set $\psi = 0.02$, so that the transmission time varies with

a standard deviation of 2% of ϕ_{ij} ¹. Note that adding noise limits the accuracy of the recalled patterns but makes the results more relevant to biological networks.

The instantaneous rate of spiking of the dendrite is computed by exponentiating the voltage $v_i(t)$:

$$r_i(t) = \gamma e^{\alpha v_i(t)}, \quad (6.6)$$

where $\alpha = 1 \text{ V}^{-1}$ and $\gamma = 1 \text{ s}^{-1}$ in this dissertation. If one assumes that observing a postsynaptic potential gives evidence in log-odds form that the dendrite should spike now, then a better way of obtaining a posterior firing rate would be to invert the log-odds by using a sigmoidal function:

$$r_i(t) = \frac{\gamma}{1 + e^{-\alpha v_i(t)}}. \quad (6.7)$$

This alternative means of generating a firing rate will be approximately equal to equation 6.6 when $v_i(t) \ll 0$. In our experiments we found this to be the case nearly all the time for a dendrite. An advantage of the sigmoidal method is that one can explicitly specify how accurately the dendrite can generate spikes, e.g. setting $\gamma = 200 \text{ s}^{-1}$ means that the dendrite cannot generate a spike with an accuracy of better than 5 ms. However we did not investigate this alternative method further, and leave it as further work.

Once the dendrite generates an event the neuron spikes and the voltage of all the neuron's dendrites are reset to a large negative voltage (v_{RESET}) which decays away in the same manner as all other synaptic inputs. This simulates the neuron's refractory period. While this violates the strict definition of a Poisson process (spikes are not independent of each other) we will still refer to the spikes being generated by a Poisson process since, conditioned on the voltage, the spikes are independent of each other. In other words, given the current voltage one can immediately calculate the instantaneous firing rate $r_i(t)$ which gives the probability of spiking in the next infinitesimal time-slice. Knowing the entire past spiking history does not provide any further information about the instantaneous rate of spiking, and in this sense the spike sequence is memoryless (or Poisson).

Informally in our model, each dendrite is a coincidence detector (Barlow, 1985). The baseline voltage can be set so that spontaneous firing happens infrequently (here

¹ In a delay line of 50ms the standard deviation will be 1ms. Almost all transmissions along this delay line will be within 2 standard deviations: (48ms, 52ms).

6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME

the baseline firing rate is once every 500 seconds). For each excitatory postsynaptic potential (EPSP) the instantaneous rate of spiking increases by a factor of $e^{\alpha w}$ for a short time (λ). If several spikes are received in a time shorter than λ , then the rate of spiking increases exponentially in the number of spikes and, for sufficiently many spikes, the dendrite will spike on a functionally-relevant timescale with high probability.

It is possible to interpret this Poisson rate of spiking as sampling from the posterior belief that a spike should occur. In this interpretation however the exponential voltage decay incorrectly handles the change in evidence over time. Exponential decay allows several computational simplifications and we leave the correct Bayesian integration of evidence as future work.

For the rest of the chapter we will use G to denote the number of spikes in a pattern. When we wire up a dendrite to detect a pattern we are free to decide which of the preceding neurons associated with the pattern to connect to. One could introduce known biological restrictions (e.g. the delays in a real biological system rarely exceed 30 ms according to Izhikevich (2006)), however we chose to connect to a random subset of the preceding neurons. Throughout this chapter we will use a default period for the patterns of 100 ms and explicitly state whenever we do not use this value. We will refer to the period by T , and mention in passing that our restrictions entail that the delay lines can be almost as long as the period ($\phi < T$). The long delay lines are biologically unrealistic, but, since precision of the delay line decreases linearly with the length of the delay, long delay lines actually reduce the overall accuracy of the network. The main reason why long delay lines were chosen was to ensure simplicity in the patterns. If a completely random pattern had an interval of complete silence for a duration longer than the maximum delay line length then this pattern could not be recalled in the network. Ensuring that no pattern had long intervals of unusual quiet would mean that the patterns are not completely random which would alter the amount of information contained in a pattern.

We denote the number of synapses in a dendrite by g , and explicitly disallow any auto-synapses, while also requiring that connections are only made to the spikes in the preceding period. This means that the total number of possible synapses is bounded above by $g \leq G - 1$. When running the simulations we also found that the performance of the network did not particularly improve for $g > 20$, yet the computational time required to run the simulation would increase substantially. For this reason we usually

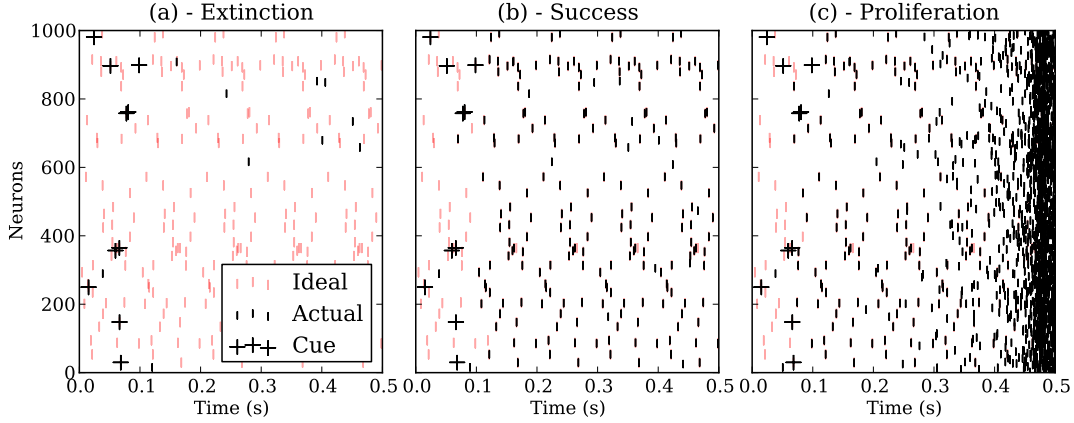


Figure 6.3: The capabilities and failures of a concurrent recall network with 1000 neurons. 500 periodic patterns (each consisting of 50 spikes and a period of 100ms) have been stored in the network. Recall is attempted by giving a starting cue of 20 spikes from one of the patterns (40% of the pattern). Panels (a)-(c) shows the result for changing synaptic strengths $w=2, 3$, and 4. Panel (a) shows the pattern dying out because synaptic strengths are too weak. In (b) there is successful recall. In (c) the network saturates because spurious spikes proliferate.

limited the number of synapses to a maximum of 20. We believe that the exact value (20) would increase if noisier dendrites were used, e.g. with higher baseline firing or larger delay noise.

In figure 6.3, 500 patterns have been stored in a network of 1000 neurons. Each pattern has $G = 50$ spikes. This means that each neuron participates in approximately 25 patterns. For every pattern that a neuron participates in, a single dendrite is added to the neuron, and hence each neuron has approximately 25 dendrites. Each pattern has a period of 100ms, and the voltage decay half-life is 5ms.

To test the recall of the network, 10 spikes are selected at random from a pattern and injected into the network. Figure 6.3 shows the three possible outcomes: either the pattern is successfully recalled, the spiking dies out, or the pattern starts to be recalled but as more spurious spikes occur the network begins to generate a proliferation of spikes and the network becomes saturated.

6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME

6.2.1 Detecting successful retrieval

Once we have wired up the network to store many patterns, we notice that there is nothing stop us recalling more than one pattern at the same time. We can give the network several partial, corrupted patterns and clean up all the patterns simultaneously, as long as proliferation or extinction does not occur (Wills, 2004). However, as shown in figure 6.4, the resulting mass of spikes makes it difficult to detect exactly which patterns were successfully recalled. We use an idea from Wills (2004) and wire simple grandmother cells to detect precise spiking patterns. Sigma-pi neurons (Wills, 2004) were wired up so that if 3 successive spikes from a particular pattern are detected with the correct temporal relationship (within a window of 3 milliseconds) then the grandmother neuron will spike. Because grandmother cells play the role of readout devices and do not affect the network dynamics, we don't add noise or background firing to the grandmother dynamics.

A sigma-pi neuron can have many dendrites, and we wire each dendrite to detect a different phase of the pattern (Wills, 2004). Thus if a pattern is correctly recalled, the associated grandmother neuron will spike steadily.

In figure 6.4, 30 patterns have been stored in a network; all the patterns have periods ranging from 30 milliseconds to 300 milliseconds. 3 patterns are selected to be recalled and 10 spikes from each pattern are injected into the network (representing 20% of the pattern). The correct response for the grandmother neurons representing those patterns is to spike regularly.

A pattern is successfully recalled if more than half the expected grandmother spikes are present. In figure 6.4 all three patterns have been successfully recalled, but we can see that the grandmother neurons hardly fired for the first period of recall. In the early stages of recall it can therefore be ambiguous as to whether the recall will be successful or not. To avoid this problem all the experiments are run for at least 10 periods to ensure that there is sufficient time for recall or failure to be unambiguous.

Having covered the neural model and discussed the detection of patterns in the network we now discuss how to evaluate the performance of the network after a noisy cue has been recalled.

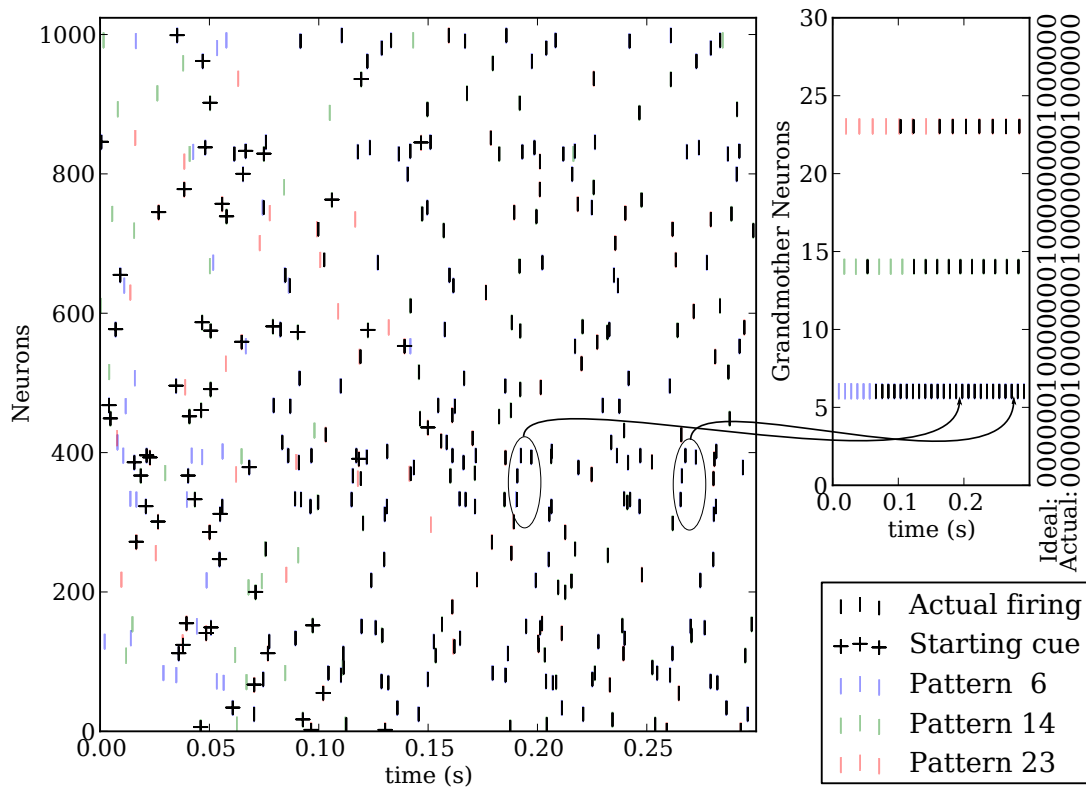


Figure 6.4: Multiple patterns can be completed simultaneously. Here we store 30 patterns and recall 3. To detect which patterns are recalled we use a grandmother neuron for each pattern. We threshold the number of grandmother spikes to estimate whether a pattern is present. This gives us the actual encoding, which we can compare with the ideal encoding.

6.3 Measuring Information

This section looks at the different types of information that can be encoded in the network. Accurate relative spike times convey a lot of information that could be used for further processing by higher-level areas. The network is also capable of recalling multiple patterns simultaneously. This allows us to measure the flexibility of the network - the number of possible subsets of stored patterns that can be simultaneously recalled.

In some sense the information capacity of a concurrent recall network (with noiseless delay lines) is infinite, as one can increase the capacity without bound by decreasing λ , decreasing v_{EQ} and increasing the synaptic strength. More specifically as $c \rightarrow 0$:

$$\lambda' \rightarrow c\lambda, \quad (6.8)$$

$$v'_{EQ} \rightarrow v_{EQ} + \log c, \quad (6.9)$$

$$w' \rightarrow w - \alpha \log c. \quad (6.10)$$

The above transformations correspond to a scaling of time, and each pattern will require ever finer precision to be recalled successfully. This means more patterns can be stored and simultaneously recalled. However in realistic physical systems infinite precision is impossible and for the rest of this paper we assume that we are unable to control λ . The noise introduced in the delays of the delay lines also provides a bound on the scaling of time, loosely speaking, if the noise in the delays (ψT) exceeds the time constant (λ) then a significant proportion of the postsynaptic potentials will have decayed away before receiving the rest of the input, and the network will cease to be as effective as the results given here suggest.

6.3.1 Information of whole patterns

One measure of information is the total number of subsets of patterns that the network can recall. The information contained in which patterns are active can be thought of as the information transmitted by the grandmother neurons (as shown in figure 6.4). We are not proposing that grandmother neurons would be found or used in a real neural architecture; rather that by measuring the grandmother neurons' output we can get an idea of how powerful a representation the network can produce.

To obtain a numerical measure of how well the network is performing we estimate the number of bits communicated (ignoring precise spike times for this section). This is the mutual information I_1 between the ideal bit vector and the recalled bit vector (as shown to the right in figure 6.4) that is created from observing the grandmother cells. If the network was perfectly reliable, then after recalling m patterns from M stored patterns $\log_2 \binom{M}{m}$ bits would have been communicated. However the network is not perfectly reliable, and some patterns might be recalled that should not be while other patterns might not be recalled that should be.

We can estimate the information contained in the set of patterns actually recalled by first estimating how many bits need to be added before we obtain the ideal encoding in which all the desired m patterns are active and all the remaining patterns are inactive. If there are m_\uparrow patterns which need to be turned on, out of a possible M_0 patterns that are off, then the total number of possible subsets is $\binom{M_0}{m_\uparrow}$, and picking one of these will require $\log_2 \binom{M_0}{m_\uparrow}$ bits. Similarly for the active patterns M_1 that require turning off m_\downarrow we incur a cost of $\log_2 \binom{M_1}{m_\downarrow}$ bits.

Subtracting the correction information that is required from the information of the ideal message, gives us the actual number of bits communicated:

$$I_1 = \log_2 \binom{M}{m} - \log_2 \binom{M_0}{m_\uparrow} - \log_2 \binom{M_1}{m_\downarrow}. \quad (6.11)$$

However this is only an estimate of the mutual information as one also needs to communicate the number of errors for both conditions (m_\uparrow and m_\downarrow). The number of bits needed to communicate this in practice is small compared to I_1 . It also requires a large number of trials to estimate the entropy of m_\uparrow and m_\downarrow accurately, so for this chapter it is ignored entirely.

6.3.2 Information of individual spike times

Another measure of information is the precise timing of individual spikes within each pattern that the network is able to reproduce. To measure this we look at the mutual information between the ideal pattern which we store, and the noisy pattern that is recalled.

To calculate the information transmitted by the neural network we imagine that we are trying to send a message of the exact times of perfectly recalled patterns. We

6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME

then compare the difference in message length between two observers. An ignorant observer would distribute probability mass equally over all possible times and neurons. An attentive observer who has seen the spikes of our network is able to place blobs of higher probability on areas where the network has spiked and smaller probability everywhere else. If the network is doing a good job of recall, then this strategy will produce shorter messages.

Whenever a spike is seen, the attentive observer then believes in a high rate of spiking (r^+) centered around the spike time, with a width 2Δ and lowers the belief in the rate of spiking in remaining areas(r^-). If n_+ spikes actually occur within the high rate of spiking and n^- within the low rate of spiking then the network has transmitted

$$I_2 = n^+ \log_2 \left(\frac{r^+}{r} \right) + n^- \log_2 \left(\frac{r^-}{r} \right), \quad (6.12)$$

bits. To calculate the information transmitted we need to know how wide Δ should be and how to distribute probability mass between r^+ and r^- . Rather than an analytic approach we search through all possible values and choose the optimal for each simulation. Technically this is cheating, as we are using the correct data to help decode the output, however on successive runs the optimal parameters change very little. This means that very little information is actually being leaked. As a thought experiment, imagine that before decoding spikes from a network, you run many similar simulations of the network and use those simulations to decide on optimal values for Δ , r^+ and r^- . These will be close to the optimal parameters for your particular decoding task, and hence your performance will be close to the optimal performance reported here.

These two different measures of information can be traded against each other and correspond to a continuum of different encoding strategies. In the next section we examine a crucial design decision, the number of patterns to store and the number that can be simultaneously recalled. We show that the encoding which achieves optimal performance for one measure is different from the other measure's optimal encoding.

6.4 Stored versus recalled patterns

When encoding information one is free to decide how many patterns to store and how many of those patterns should be simultaneously recalled. If many patterns are stored then fewer can be simultaneously recalled before proliferation occurs.

6.4 Stored versus recalled patterns

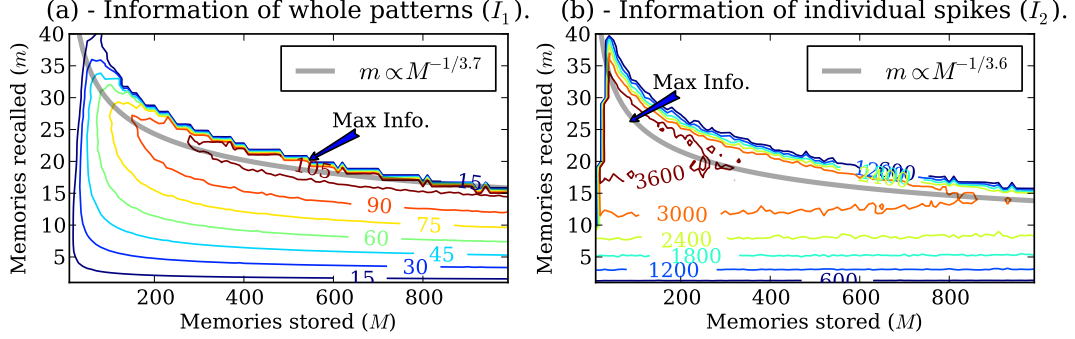


Figure 6.5: The number of bits per period (shown in contours) that can be communicated in the network as a function of the number of patterns that have been stored (M) and the number of patterns simultaneously recalled (m). For a given number of stored patterns the maximum information is approximated by $m \propto M^{\frac{1}{D-1}}$ for both information measures.

Wills (2004) analyses a model in which a dendrite will fire if D spikes are received in a short time period. His analysis predicts that the maximum number of patterns that can be recalled (m) as a function of the number of patterns stored (M) is

$$m \propto M^{-\frac{1}{D-1}}. \quad (6.13)$$

Since our neural model is similar, one might expect the same conclusion to hold here. In figure 6.5 we show contour plots of both measures I_1 and I_2 for different numbers of patterns stored and then recalled. We fit curves to the maximum performance for each level of stored memories and there is a good fit. To start each selected pattern, we provide an initial cue containing half the spikes of the pattern's spikes. The network is left to run for 1 second (10 periods of recall). If proliferation is detected, then there is no information transmitted (proliferation is defined as twice the number of expected spikes). It is important to note that figure 6.5 also fixes the number of spikes in a pattern $G = 50$ and the number of synapses in a dendrite $g = 20$. In section 6.6.2 we explore the effect of changing the pattern size. We were careful to choose the synaptic strength for unbiased spike times as will be detailed in the next section. For a different synaptic strength the shape and performance shown would also change, but the general observations remain valid.

We summarise the optimal configurations for both measures below:

6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME

- **I_1 optimal** : Figure 6.5(a) shows that I_1 is maximized when many patterns are stored. Precise recall is not important, as long as the patterns are recalled sufficiently accurately that they are detected by grandmother neurons. As one approaches the proliferation boundary there are more spurious spikes, but these do not affect I_1 , so the optimal performance is close to the boundary.
- **I_2 optimal** : Figure 6.5(b) shows that I_2 is roughly independent of the number of patterns stored (M). Instead the information only depends on the number of patterns being recalled (m). Since the proliferation boundary satisfies $m \propto M^{-\frac{1}{D-1}}$, we can recall more patterns simultaneously by only storing comparatively few patterns. Note that the fitted line of best performance is not on the edge of the proliferation boundary as spurious spikes begin to appear near the boundary, which reduce the information contained in the spike times.

After having decided on how many patterns to store and recall one can maximise the network's performance by ensuring that the expected spike times are unbiased. One can do this by tweaking the synaptic strength w and slightly adjusting the delay lines. We cover this in the next section.

6.5 Synaptic strength and transmission delays

After having chosen the number of patterns to store and simultaneously recall, the synaptic strength (w) should be as large as possible while avoiding proliferation. This allows the initial cues to be as small as possible, while still ensuring successful recall. However in figure 6.6(a) we can see that setting the synaptic strength high means that the dendrites will be biased to spike early and there will be a phase precession. (For illustrative purposes we ignore the discrete nature of individual spikes in figure 6.6(a).) This bias will affect the information measured by I_2 as the spike times are not as precise as is possible.

If the network is biased to spike early then the entire pattern will be sped up. There is some evidence that recall in dreaming animals is faster than in real life (Ji & Wilson, 2006), which suggests that our definition of I_2 should be invariant to scaling. A similar effect can be achieved simply by scaling the delay lines to counteract the bias, which is the approach we take here. The delays can all be scaled by a constant factor to

6.5 Synaptic strength and transmission delays

ensure that the average recall time is unbiased. The optimal scaling for a pattern with $G = 50$ spikes and $g = 20$ synapses per dendrite is shown in figure 6.6(b). To find the empirically-best scaling time we stored and recalled a single pattern while looking for drift in the timing of the pattern. Other experiments (not shown here) confirmed that the optimal strength appears insensitive to the number of patterns simultaneously recalled.

We found the scaling for our synaptic strength could also depend on the introduced noise in the delay lines (ψ). If there is a large variation in the arrival times of post-synaptic potentials then it is possible for a particularly early postsynaptic potential to have decayed away before the correct time to spike. This early arrival then does not contribute meaningfully to the correct timing of the postsynaptic spike and as a result more synapses (or stronger synapses) are required. To ensure that this different regime of synaptic scaling is avoided we ensure that two standard deviations of the noise in the longest possible delay line is still less than the time constant of the decay:

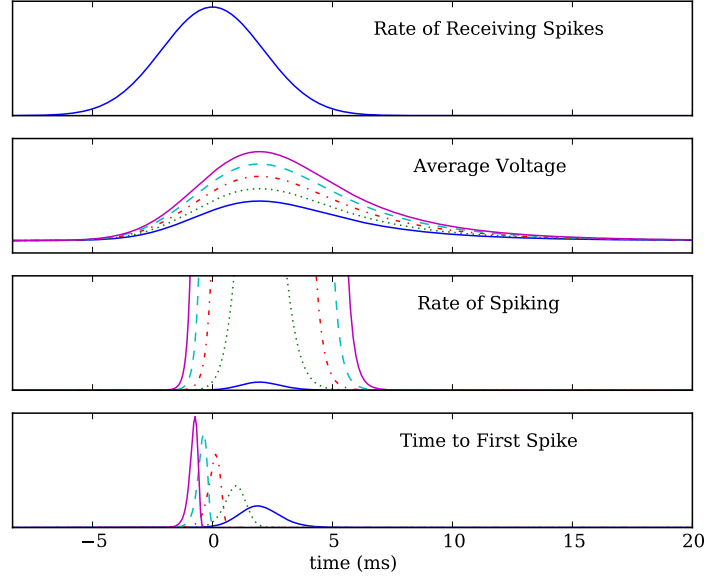
$$2\sigma < \lambda. \quad (6.14)$$

In our simulations, $\sigma = 0.02T = 0.002\text{ s}$ and $\lambda = \frac{0.005}{\ln 2}\text{ s}$, so this requirement is satisfied. Later in figure 6.9(a) we examine the effect of changing T and see reduced spike accuracy beyond this limit.

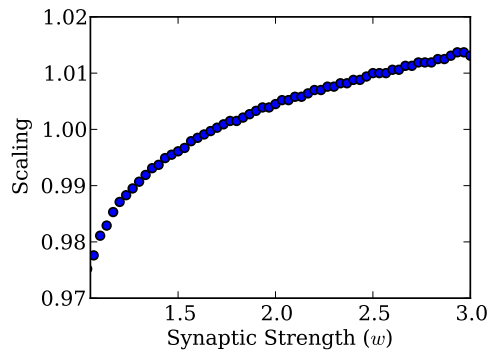
Later in the chapter we will look at changing the pattern size G and the number of synapses per dendrite g . However these changes also affect the ideal synaptic strength. We need to ensure that our results are not affected by this complication. We explored this in figure 6.6(c) by storing and recalling a single pattern in the network. We varied the synaptic strength w and the group size G (setting the $g = G - 1$) and looked for the pattern with minimal drift (after ten periods of firing). The best matching synaptic strengths all lie on the line $w \propto \frac{1}{G}$. This suggests that only the total synaptic input matters.

We have now covered how to optimally wire up a network given the number of patterns to store and the number required to be simultaneously recalled (for patterns of a fixed size). In the next section we quantify the two measures of performance for the network when performing recall given a noisy cue. We will look at the level of noise in the cue, as well as the effect that pattern size (G) has on both measures I_1 and I_2 .

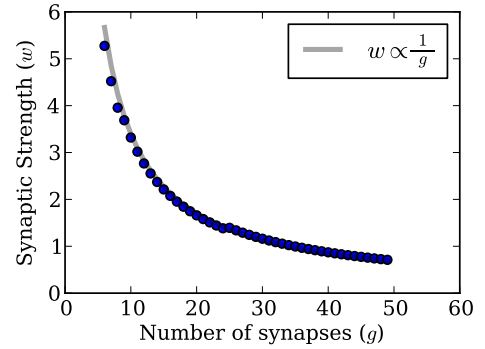
6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME



(a) Synaptic strength can introduce bias in the spike time. Here we assume that the arriving spikes are distributed around the correct time ($t = 0$) but have intrinsic noise. The average voltage will depend on the synaptic strength, and we plot several different averages for different synaptic strengths. This in turn affects the average rate of spiking and hence the distribution of the time to the first spike.



(b) The empirically-observed scaling for unbiased spike times at a given synaptic strength.



(c) The optimal strength as a function of group size. The relationship is well approximated by $w \propto \frac{1}{g}$ which suggests the total synaptic input ($w \times g$) is constant.

Figure 6.6: Correctly setting the synaptic strength: (a) synaptic strength can introduce a bias to spike early or late, but can be corrected for by scaling the delay lines in (b). In (c) as the pattern size (G) changes the ideal synaptic strength scales in such a way as to suggest that the ideal total synaptic input remains constant.

6.6 Information Retrieval and Pattern Size

For the network to be useful we need to show that the network can improve an initial noisy cue which has jittered and missing spikes. We will repeat our experiments on pattern completion under two different conditions, firstly when storing a few large patterns and secondly when storing many small patterns. This provides insight into some of the tradeoffs of group size when cleaning up a noisy cue.

To construct the network with large patterns, we store $M = 10$ patterns of $G = 500$ spikes each, in a network of 1000 neurons. Each dendrite has $g = 20$ synapses which each connect to a preceding spike. We start by measuring the information in an initial set of spikes (the noisy cue). For the large patterns only a single pattern is recalled and the initial set is constructed by selecting a number (n) of the spikes to activate. We jitter these initial spikes using a uniform distribution with a width 2Δ centered on the correct time. These initial spikes contain information about the underlying correct patterns and we show the information contained (as a function of n and Δ) in figure 6.7(a).

The set of spikes in the initial cue is fed into the network and run for 10 periods. We then measure I_2 for the network's set of spikes in the final period of recall (plotted in the middle column). There is a clear boundary between successful recall and extinction in figure 6.7(c).

The difference in information between the initial cue and the final set of spikes represents the ability of the network to recall information. This difference is shown in figure 6.7(e). In spite of the noise inherent in the network (i.e. spontaneous spikes and transmission noise) there is still a large positive region. This represents the network cleaning up the signal, by providing lower noise and ensuring more spikes are present.

The noisy dendrites account for the negative region in the lower right of figure 6.7(e). Here the initial cue has many spikes with high accuracy. The network is unable to reproduce such a high quality signal which results in a loss of information. With better hardware (less noise in the delay lines, smaller time constant for voltage decay and a lower rate of spontaneous spiking) the network would be able to reproduce a higher quality signal and more information would be communicated.

For the second row of figure 6.7 we examine the performance of a network with small patterns. We now store 10 times as many patterns ($M = 100$) but each pattern

6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME

contains 10 times fewer spikes ($G = 50$). This ensures the total number of dendrites is constant and the number of synapses is kept constant too. We now select 10 patterns to activate with a comparable number of starting spikes (i.e. the cue has a comparable amount of information – compare figures 6.7(a) and 6.7(b).

When starting 10 patterns there is a broader transition between completely successful recall and completely unsuccessful recall as shown in figure 6.7(d). This is because it is possible for several of the patterns to be unsuccessfully recalled, which is not possible when only a single large pattern is being recalled. To ensure all patterns are reliably recalled we need a better cue, which reduces the maximum I_2 -performance of the network. In figure 6.7(e) the network is capable of providing approximately 2250 bits of information over the initial starting cue, however, when recalling 10 small patterns the network can only manage 1900 bits. This reduction is in part also due to increased spike-time jitter in small patterns. The typical successful recall of a large pattern results in approximately 3000 bits of information. However successfully recalling 10 smaller patterns typically results in 2750 bits of information. We believe smaller patterns are affected more by individual errant spikes.

While there is a reduction in the maximum performance when measured in individual spike times, there is an increase in the information contained in the state of the network. With 10 large patterns the network can only recall one of them, hence it can only represent 10 states and will only transmit $\log_2 10 \approx 3.3$ bits. However with more numerous smaller patterns the network can recall any 10 of 100 patterns, and it can represent $\binom{100}{10}$ states which transmits $\log_2 \binom{100}{10} \approx 43$ bits. We will further explore the effect that different pattern sizes have on both I_1 and I_2 in section 6.6.2.

It is also useful to know the form of the least informative cue that still results in successful recall. We are able to characterise the boundary layer between extinction and successful pattern recall (this is plotted in a grey line in figure 6.7(c)). Since we have an analytic form for the information contained in an initial cue we can then find the cue with minimal information that still results in successful recall. This is shown in figure 6.7(e) and agrees closely with the empirically optimal point.

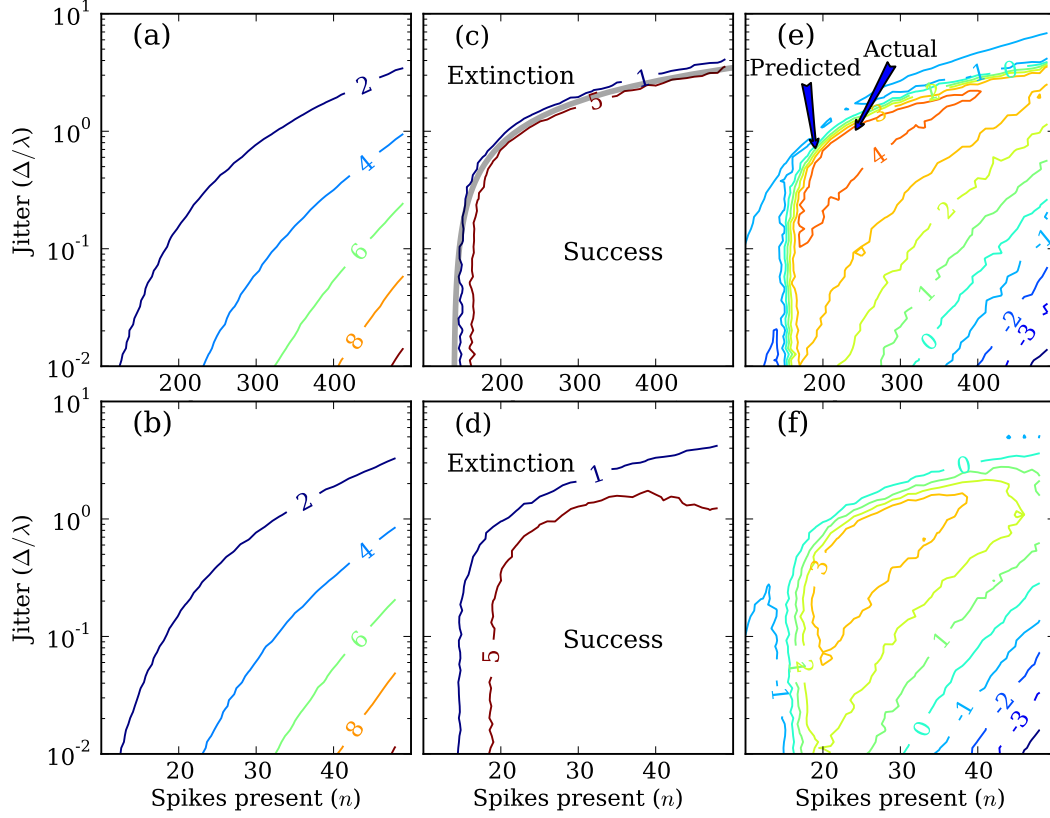


Figure 6.7: The contrast in spike-time information (I_2) between the recall of one large pattern of 500 spikes (top row), versus recalling 10 small patterns of 50 spikes each (bottom row). The information is a function of n (the number of spikes in the initial cue) and Δ (the amount by which the spikes are jittered). (a)–(b) The information in a given cue before–, and (c)–(d) after– giving it to the network. The difference between initial and final information is shown in (e)–(f). The contours are labeled in units of bits per expected spike. There are 500 expected spikes so if the network provides of 4 bits per spike then the network is providing 2000 bits of total information. The predicted recall boundary is plotted in light grey in (c) and the predicted maximum information is shown in (e), both agree well with the empirical results.

6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME

6.6.1 Maximum I_2 recall performance

Here we sketch a derivation for the boundary of successful recall with the intention of finding the minimally informative cue which can be successfully recalled. We express the boundary as a function of n the number of spikes in the cue and Δ the width about which the spikes are jittered. For a dendrite to successfully spike within λ of the correct time requires that the voltage be driven approximately to:

$$v_{\text{crit}} = -\alpha^{-1} \log(\gamma\lambda), \quad (6.15)$$

where α and γ are defined in equation 6.6 on page 119. We start by approximating discrete spikes with a continuous input. After time Δ the dendrite will receive an average continuous input of $\left(\frac{ngw}{G\Delta}\right) \text{Vs}^{-1}$, where $\frac{n}{G}$ is the proportion of activated synapses, g is the number of synapses in the dendrite and $\frac{w}{\Delta}$ is the intensity of incoming voltage. Starting from equilibrium the voltage behaves as:

$$v(t) = \left(\frac{ngw}{G\Delta}\right) \lambda \left(1 - e^{-\frac{t}{\lambda}}\right) + v_{\text{EQ}}. \quad (6.16)$$

Setting $v(\Delta) = v_{\text{crit}}$, we can obtain the relationship between Δ and n :

$$n = \frac{G(v_{\text{crit}} - v_{\text{EQ}}) \left(\frac{\Delta}{\lambda}\right)}{gw \left(1 - e^{-\frac{\Delta}{\lambda}}\right)}. \quad (6.17)$$

This boundary has been plotted in figure 6.7(b) as a thick grey line and lies almost precisely over the boundary.

Now that we know the boundary of successful recall we can substitute it into the analytical formula for the information in a noisy cue. Finding the cue that contains the smallest amount of information but is still successfully recalled means the network is filling in the most information and is operating at its most efficient.

If n spikes are present and have a jitter of width Δ then the information (from equation 6.12 on page 126) is:

$$I_2 = n \log_2 \left(\frac{r^+}{r}\right) + (G - n) \log_2 \left(\frac{r^-}{r}\right) \quad (6.18)$$

The missing spikes could occur at any time so that the low rate of spiking is given by:

$$r^- = \frac{G - n}{NT}. \quad (6.19)$$

The jittered spikes each occur within a window Δ wide and will also contain some of the missing spikes, which makes the high rate of spiking:

$$r^+ = \frac{1}{\Delta} + \frac{G - n}{NT}. \quad (6.20)$$

The completely ignorant rate is :

$$r = \frac{G}{NT}. \quad (6.21)$$

Substituting the above equations and equation 6.17 into equation 6.18, and differentiating with respect to Δ we obtain a lengthy equation which we are able to solve numerically. This maximum is plotted in figure 6.7(e) and is close to the empirical maximum.

We now explore the effect that changing the pattern size has on both I_1 and I_2 , after which we explore how the information scales as we scale the number of neurons N or the period of the patterns T .

6.6.2 Pattern Size

The trade-off between I_1 and I_2 for patterns of different sizes is clearly seen in figure 6.8. Here we created many nets with patterns of different sizes (G). For a given G we search all combinations of patterns stored (M) and simultaneously recalled (m), as shown in figure 6.5. We plot the maximum of $I_1(G)$ in figure 6.8(a) and the maximum of $I_2(G)$ in 6.8(b). We set the number of synapses on a dendrite $g = \min(G - 1, 20)$ as the performance did not appear to significantly improve with more synapses, yet the computational costs for $g > 20$ were significant.

The performance for spike-time accuracy I_2 increases with pattern size G . (The increasing variation for large G is as a result of having to use a discrete number of large patterns.) The maximum possible spike-time accuracy would occur in the limit of an infinitely-large group of spikes stored in an infinitely-large network. This infinitely-large network would still not achieve arbitrary precision, but would be limited by the equilibrium voltage as well as the inherent noise in the delay lines. The infinitely-large pattern would not be susceptible to drift however, which does affect smaller patterns.

While the information in individual spike times keeps increasing with group size, the information for whole groups of spikes (I_1) peaks for relatively small groups and

6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME

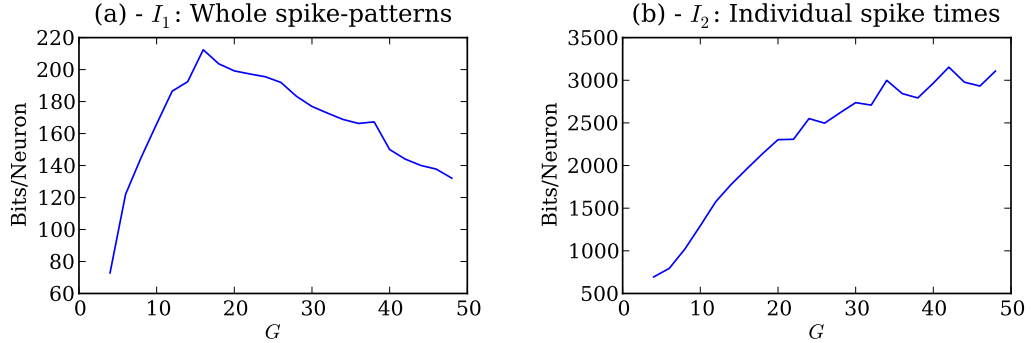


Figure 6.8: Optimal capacity of the network as a function of the number of spikes per pattern G .

then declines as shown in figure 6.8(a). This is as a result of two opposing forces. Small patterns are more likely to turn on spontaneously or fail to turn on correctly; as pattern sizes get larger they also get more reliable. However patterns that are too large begin to negatively affect the total number of patterns that can be stored and recalled. The ideal pattern size will be a function of the equilibrium voltage and the noise in the delay lines, but we did not investigate this further.

6.6.3 Linear Capacity

In figure 6.9 the capacity of a network is plotted as a function of the period. By doubling the period, the network is able to store approximately double the number of patterns and simultaneously recall double the number of patterns as well. This results in a doubling of the capacity. Similarly if the number of neurons is doubled then twice as many patterns can be stored and recalled.

A linear relationship means each spike carries the same amount of information regardless of network size or period. This is much better than a rate encoding. In rate encoding the number of spikes a neuron produces in a given time is the quantity of interest. However if the neurons are noisy then there is some uncertainty regarding the intended number of spikes as opposed to the observed number of spikes. If one wants to halve the uncertainty in the current spiking rate then one requires a four-fold increase in the required time. This is a result of the standard deviation of the

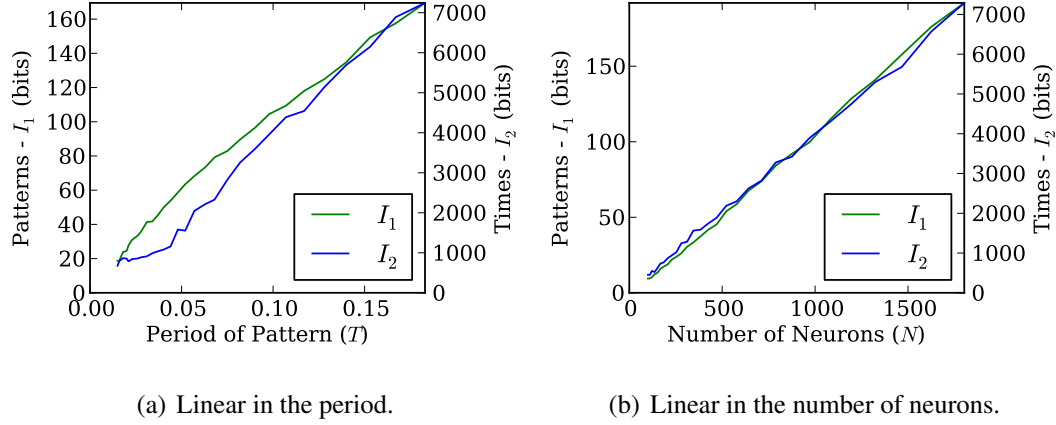


Figure 6.9: For both measures of information I_1 and I_2 , the capacity of the network is approximately linear in both period and neurons

rate estimate growing as $O(\frac{1}{\sqrt{n}})$ (Gerstner *et al.*, 1996). Naïve population codes that function as parallel rate codes are similarly inefficient.

Note that the linear capacity only holds approximately for the period (T). As T increases the transmission noise also increases and equation 6.14 no longer holds. This means the dendritic voltage decays significantly between the first presynaptic spike and the ideal spike time. Accurate spiking then requires significantly more presynaptic spikes and the relative performance will degrade.

6.7 Discussion

This chapter looked at extending an associative memory into the continuous-time domain. In doing so we arrived at a network which had many elements in common with a biological neural network. This led us to adopt neural terminology, in spite of some aspects which still need further work before the entire network is biologically plausible.

In continuous time the random patterns that we would like to store are samples from a Poisson process. Given a small noisy sample of the pattern as the initial cue we would like to reduce the noise and fill in the rest of the pattern.

6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME

We achieved this by using delay lines which were calibrated so that the postsynaptic potentials from the preceding spikes in the pattern would all arrive simultaneously at a dendrite. We then used a neural model which would detect coincident spikes and used each dendrite to detect a single pattern. To store a pattern in the network we would explicitly add a dendrite to each neuron that spiked in the pattern. While adding dendrites to the network is biologically implausible, a similar effect could be achieved in a neurally plausible manner through the activation of dendrites which are already present and have pre-existing weak synapses and delay lines. The synapses with matching delays could then be strengthened when a new pattern is stored.

To measure the performance of the network one can look at how accurately the patterns are recalled. There is a large amount of information contained in accurate spike times, and we looked to transmit information in accurate spike times with an inherently noisy neural model. Accurate spike timing is not the only source of information that the network can communicate. The design of this network also allows many patterns to be recalled simultaneously. The network can communicate information through the choice of a particular subset of patterns to recall out of all possible subsets of patterns that it could recall. We called the information contained in the particular subset of active patterns I_1 and the information in accurate spike times I_2 .

The network requires many parameters to be set, and it is not immediately clear what effect these parameters have on the performance of I_1 and I_2 . This chapter looked to elucidate some of the important relationships, such as how many spikes each pattern should contain, and how many patterns to store and how many patterns to simultaneously recall.

6.7.1 Future work

As future work, the main aspect in which performance could be improved would be to use a principled probabilistic detection of the patterns. This would entail designing a generative model of the cueing of patterns and using Bayes theorem to detect the patterns. We expect that several improvements would be possible: the main improvement would be the principled setting of the synaptic weights. The spiking of neurons that participate in very few patterns, would provide stronger evidence for the presence of a pattern when it does spike. Also, the standard deviation of the delay line noise varies

proportionally to the length of the delay line and we would expect the evidence from observing a postsynaptic potential from a short delay line to be strong, but short-lived. Conversely the evidence from the postsynaptic potential of a long delay line would be weak, but long-lived. For computational reasons in this chapter we instead chose to give each postsynaptic potential equal weight and duration.

Adding hardware whenever a new pattern is stored is neither particularly elegant (nor particularly plausible). For future work we would prefer to follow the approach taken throughout most of this dissertation where hardware isn't specifically added, but the state of the hardware is changed in response to the storage of a pattern. Since the network is already close to being neurally plausible, we might seek solutions that are also neurally-plausible, and this would take the form of strengthening synaptic weights (Bliss & Lømo, 1973). It is also worth mentioning Wilson (2009), which increases the I_1 capacity by effectively embedding a Hopfield network in each pattern. This embedding allows each recalled pattern to be in one of many different states.

Providing a maximum firing rate for any dendrite (as shown in equation 6.7), is also a natural extension to explore. Using this formulation then regardless of the evidence a neuron will be unable to spike with an accuracy that can be specified in advance. While the information contained in accurate spike times formed a major thrust of this chapter, we believe that useful research will come out of accurate spike-time coding using cheap and cheerful hardware that is intrinsically noisy.

Some of the delay lines we consider are implausibly long (some delays will be nearly 100ms) yet are sometimes found in biological neural systems (Lee *et al.*, 1986) (Miller, 1987). As future work we could add biological plausibility by eliminating these long connections, and see if accurate spike timing in these patterns is maintained. We would predict the accuracy to remain the same, or even improve, since the standard deviation of the delay line noise is not allowed to grow, but is instead averaged out over many short intervals. This would however introduce another restriction on the number of spikes needed for patterns with long periodicity.

Enforcing shorter delay lines would also enable successful recall for smaller cues. Rather than randomly selecting spikes from a pattern to present as a cue, we could present the initial part of the pattern, and the network would recall the rest. This is an option which is briefly explored in Wills (2004).

6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME

In Faisal & Laughlin (2007) there is a detailed discussion of the types of noise that can affect transmission times. There are many nonlinear effects observed, but biological neurons can deal with transmission time noise on the order of 10%, as opposed to the 2% used here. In part this extra noise could be averaged out by using larger patterns, but interesting further work would be to investigate the optimal coding strategies in increasing noise conditions. We believe there would be a natural progression from a spike-time code in low noise conditions to a code which is more rate-like in high noise conditions.

Delay lines that are very noisy are equivalent to delay lines that are specified with low precision. Increasing noise conditions can then be thought of as trying to find short descriptions of the network's delay lines such that it is still able to perform accurate recall. This is not quite equivalent to finding a short description length of how to construct the system but it does mean that the system is able to function with components that do not require great precision.

As a possible direction for future work we have also briefly explored the effects of inhibitory neurons which were wired up randomly in the network, and helped avoid network proliferation. Initial experiments suggested that I_1 could be doubled in this manner, although the random inhibition of spikes does negatively affect I_2 . If we suspected that not all spikes in a pattern needed to be present for a downstream area to extract all the necessary information then we could conceivably modify I_2 such that the information about spike timing is contained in the most accurate 50% of the spikes. Inhibition would not affect this newly modified measure I_2 , and it seems plausible that the new network would be capable of more computation than the old.

6.7.2 Conclusions

It is exciting that multiple patterns can be simultaneously recalled, in an architecture that was originally designed to only recall single patterns. This allowed us to code information in the network in two ways: we could transmit information by specifying which particular subset of patterns was active out of all possible subsets (I_1), we could transmit information through the accurate timing of spikes relative to each other (I_2).

If the task our network is trying to solve requires information coded in I_1 then it is best to store lots of patterns and recall only a small number of them. Each pat-

tern should consist of as small a number of spikes as is possible while still remaining reliable.

If the task requires information in I_2 then we can produce accurate spike times by using large patterns, storing a small number of these patterns but recalling a far higher proportion of the patterns. With these parameters the network will provide lots of accurately-timed spikes from which information can be derived.

If we want to recall patterns using a cue with only a small number of spikes, then we need to have strong synapses so that only a few presynaptic spikes will ensure reliable postsynaptic spiking. This places a limit on the number of patterns we can store. Patterns that use strong synapses for recall also tend to be biased and are recalled faster than they should be. A simple solution to avoid this bias was to scale each delay line by a constant factor so that recall is still unbiased.

It should also be noted that although this chapter deals exclusively with periodic patterns, there is nothing to stop the same concepts being applied in a layered architecture where a noisy pattern is received by a lower layer, and partially cleaned up as it is passed to a higher layer.

This chapter has shown that encoding information in continuous-time recall can be achieved in a variety of ways and the optimal coding strategy will depend very much on the exact task. This flexibility suggests that many different tasks could be achieved using such an encoding and hence the network merits further study.

6. ASSOCIATIVE MEMORY IN CONTINUOUS TIME

CHAPTER 7

Conclusions

This dissertation has considered solving the associative memory task using distributed Bayesian inference. Under this setting we get to observe many patterns $\{x_r\}$ one at a time, and update the storage z after seeing each pattern to reflect the new set of stored patterns. In this thesis we calculated independent functions for each bit (or integer) in z and used the logical OR operator or simple addition to store the results for each new pattern. This created a distributed representation and we were particularly interested in creating systems with short description length. If we had to explicitly describe the connectivity for every element of storage in z then our system would not have a short description length. This led us to consider creating systems where, instead of an exact specification, we can simply use random connectivity.

Bayesian inference allows us to use the information contained in z to calculate what the pattern should be (Sommer & Dayan, 1998):

$$\Pr(x | z) \propto \Pr(x) \Pr(z | x), \quad (7.1)$$

where $\Pr(x)$ refers to our prior belief over the pattern. In this dissertation we referred to our prior belief as the cue, and to highlight this we used $\Pr(c)$. Principled statistical decoding has several advantages. If our model of the noise is correct then this is the optimal decoding method. Principled decoding does not suffer from catastrophic failure when the associative memory is overloaded, but instead the performance gracefully degrades back to the prior cue. This dissertation restricts ourselves to simple functions which allow us to solve the inference task with message-passing techniques.

7. CONCLUSIONS

7.1 Overview

This thesis presented a neurally-inspired generalization of the Bloom filter in which each bit is calculated independently of all others. This independence allows computation to be carried out in parallel. The independence also allows the network to be robust in the face of hardware failure, allowing it to operate with an optimal probability of error for the degraded storage.

However, the neurally-inspired Bloom filter required extra storage to achieve the same probability of a false positive as a traditional Bloom filter. It also has a higher rate of errors for items that are similar to already-stored items. In this sense it is not a good Bloom filter, but it can be turned into a great associative memory. This modification allows a noisy cue to be cleaned up with a small probability of error. We also showed the performance of the network can be measured as the improvement it is able to make in a noisy cue. Our neurally-inspired associative memory has more than twice the efficiency of a Hopfield network (even if the Hopfield network is decoded optimally).

To assess the performance of an associative memory one needs to know the probability of bit error in the associative memory for a given number of stored patterns. However, if one is given more storage then it is unclear whether one should try to store more patterns, or reduce the error rate for the current number of patterns. To solve this problem, we found a natural definition of the capacity of an associative memory. The formula is based on information-theory results for the binary symmetric channel. The capacity did not require an arbitrary probability of error as other definitions of capacity do. There are many problems associated with measures of the capacity of an associative memory that are do not use results from information theory (Knoblauch *et al.*, 2010).

Modifying the binary associative memory allowed us to create an associative memory which gradually forgot old patterns (known as a palimpsest memory). It did so by overwriting old patterns with new patterns. However the old patterns were overwritten in a haphazard fashion with bits being overwritten independently. The capacity of the palimpsest memory was much smaller than the associative memory that inspired it. This suggested that overwriting representations independently is bad, and we explored two possible means of avoiding this; using integer storage we could superpose representations, or we could overwrite the bits in a dependent fashion.

We considered superposing representations by allowing integer storage which meant we could add representations together. This naturally led us to consider one of the most well-known networks – the Hopfield network. We compared the performance of the traditional Hopfield decoding rule with the performance of several principled statistical decoding rules, and found that being principled improves the performance, often with no increase in the required computations. We also considered sparse encoding rules, which improved the capacity of the memory and provided evidence that sparse representations are more efficient.

Using ideas from error-correcting codes we designed an associative memory that avoided overwriting bits independently by grouping many computational units together into small populations, and storing one pattern per population. This associative memory was more complicated than the others previously considered as there are several stages to recall; we must first identify the correct population and then we use a simple error-correcting code to clean up the cue. This associative memory is very efficient for large patterns which must be recalled using cues with low levels of noise.

Finally we considered extending the associative memory task into continuous time. This produced a network which was very similar to a biological neural network and we freely adopted terms from biology. Instead of storing binary vectors we now considered storing samples from a Poisson process which can be thought of as neural spikes. To recall a pattern each spike was allocated a dendrite in the associated neuron. The dendrite was wired up with delay lines to preceding spikes in the pattern. These delay lines were imprecise and noise was added to each transmission. Each time a dendrite received a post-synaptic potential this could be viewed as evidence for the presence of the pattern. Strictly speaking, the evidence for a spike would vary in both strength and duration depending on many factors. As a computational shortcut we instead chose to use a single weight and duration for all post-synaptic potentials. We showed that this network could recall many patterns simultaneously which allowed information to be coded in both the accurate spike times that the network produced as well as the particular set of patterns that were being recalled.

7. CONCLUSIONS

7.2 Discussion

Nature often finds efficient solutions, yet specifying exact connectivity for a brain of M neurons will potentially require an $O(M^2)$ description length. If we are interested in building brain-like devices, then we should also be interested in devices that can be constructed from a short description (e.g. a genome) yet still perform within a constant factor of optimal performance. These considerations would not be important when building small brains, since a description of the entire connectivity is short compared to the genome (indeed it appears that the entire synaptic connectivity of *C. Elegans* is completely specified (White *et al.*, 1986)). However for large brains, the description would grow rapidly. It is also possible that there is a short description that specifies implicit constraints, without giving explicit details on how to satisfy those constraints. This is a possibility that we have not explored at all in this dissertation, but note that the spontaneous activity in the developing brain could be finding a solution which satisfies those constraints.

Using functions of randomly chosen bits ensures that the description of the system is short. When describing the system we are able to simply say that the connections are random, rather than exactly specifying the individual connections. Completely random connectivity is the ultimate in short description length, it is the distribution with the maximum entropy and one need not describe *any* of the actual connections. In a real biological system we would expect completely random connectivity would be difficult to achieve in the face of spatial constraints, but posit that systems with short descriptions are interesting as they represent systems that might be easy to evolve.

We could also improve the performance of the network by considering connectivity other than purely random. There is work (Vertes, 2010) which suggests that small-world connectivity performs far better than completely random connectivity when creating continuous-time associative memories of the form in chapter 6.

This dissertation has focused on systems that can be constructed using a short description. Another related direction for future work is to consider systems whose *operation* does not require fine precision. We touched on this briefly when we added transmission noise to the delay lines of the continuous-time network. Analogously we could consider low-precision messages in the discrete networks. There is already a

large literature for such a direction from the field of error-correcting codes (Richardson & Urbanke, 2001).

An important trade-off that has not been considered here is trading efficiency for speed of computation. For all the solutions described here, successful recall requires far less iterations when the number of patterns stored is far less than the optimal number. In real animals there would be selection forces favouring animals with efficient brains using the smallest number of neurons for their niche, since neurons are metabolically expensive tissue. However the disadvantages of an inefficient brain (which uses more metabolically-expensive neurons than it needs to) could be offset by increased processing speed, or faster learning. However we leave this as future research.

We have also shown several points in the dissertation where we can avoid analysis of the inference algorithm and instead focus on the inherent probabilities of failure. Examples of this include: our theoretical model of the probability of failure in the neurally-inspired associative memory (on page 40) as well as the scaling for the minimum connectivity as a function of cue-noise in the cued-error-correction task (on page 96). In both of these cases we were able to obtain insight by assuming that the inference worked perfectly and then focusing on other hard constraints.

The graphs that we performed inference in were incredibly loopy, yet inference was efficient and accurate. This is due to the nature of the functions we were considering. The messages associated with the conjunction of many variables are only informative if we believe that nearly all the variables are true. The messages associated with the parity code are similarly uninformative if there are at least a couple of bits we are uncertain of. This works as an implicit sparsifier, and if one considers only the informative messages then the effective graph is far less loopy.

Massively-parallel, fault-tolerant associative memory is a task that the brain solves effortlessly and represents a key task we still do not fully understand. While the work presented here uses simple computational elements that are crude in comparison to biological neurons, we hope that it provides insight into how such memories might be created in the brain.

7. CONCLUSIONS

APPENDIX A

Commonly Used Symbols:

Symbol	Description
x	Binary input vector
z	Binary storage (and associated processing)
N	Dimensionality of x
M	Dimensionality of z
R	Number of patterns to be stored
k	Number of hash functions used in a Bloom filter
a	Number of “AND”s in a function
b	Number of “OR”s in a function
h_m	m^{th} hash function
f_m	m^{th} encoding function
g_m	m^{th} index ranging from $1 \dots N$
$s(p)$	Signal performance (in bits): $s(p) = 1 - H_2(p)$
p_x	Probability of bit error in x after recall
p_z	Probability of bit error in z
c_n	Probability of a bit in the cue being on $\Pr(\hat{x}_n) = c_n$

A. COMMONLY USED SYMBOLS:

APPENDIX B

Commonly Used Symbols: Associative Memory in Continuous Time

Symbol	Default	Description
G	50	Number of spikes in a Group (a.k.a. pattern, memory)
g	20	Number of synapses in a dendrite ($< G$ as auto-synapses are not allowed)
M		Number of Memories stored in the network
m		The number of patterns actively recalled
N	1000	Number of Neurons
w	2.0 V	Synaptic strength - when a spike is received the voltage of the dendrite is increased by s
T	0.1 s	Period of a repeating memory (also represents the maximum possible delay in synaptic transmission)
λ	0.007 ms	Time constant of voltage decay
σ	0.02ϕ	Standard Deviation of delays
Δ		Maximum jitter time
ϕ_{ij}		Time delay of the j th synapse on the i th dendrite
v_{RESET}	-40	The voltage a dendrite is reset to immediately after a spike
v_{EQ}	$-\ln 500$	The resting voltage at equilibrium

B. COMMONLY USED SYMBOLS:
ASSOCIATIVE MEMORY IN CONTINUOUS TIME

References

- ABELES, M. (1991). *Corticonics: Neural circuits of the cerebral cortex*. Cambridge University Press. 112
- ACKLEY, D., HINTON, G. & SEJNOWSKI, T. (1985). A learning algorithm for Boltzmann machines. *Cognitive Science: A Multidisciplinary Journal*, **9**, 147–169. 69
- ALEKSANDER, I., THOMAS, W. & BOWDEN, P. (1984). WISARD, a radical step forward in image recognition. *Sensor Review*, **4**, 120–4. 24
- ALMEIDA, P., BAQUERO, C., PREGUIÇA, N. & HUTCHISON, D. (2007). Scalable bloom filters. *Information Processing Letters*, **101**, 255–261. 19
- AMIT, D. & FUSI, S. (1994). Learning in neural networks with material synapses. *Neural Computation*, **6**, 957–982. 49, 56, 57
- AMIT, D., GUTFREUND, H. & SOMPOLINSKY, H. (1985). Spin-glass models of neural networks. *Physical Review A*, **32**, 1007–1018. 66
- AVIEL, Y., PAVLOV, E., ABELES, M. & HORN, D. (2002). Synfire chain in a balanced network. *Neurocomputing*, **44**, 285 – 292. 111
- AYZENSHTAT, I., MEIROVITZ, E., EDELMAN, H., WERNER-REISS, U., BIENENSTOCK, E., ABELES, M. & SLOVIN, H. (2010). Precise Spatiotemporal Patterns among Visual Cortical Areas and Their Relation to Visual Stimulus Processing. *Journal of Neuroscience*, **30**, 11232. 113
- BARBOUR, B., BRUNEL, N., HAKIM, V. & NADAL, J. (2007). What can we learn from synaptic weight distributions? *TRENDS in Neurosciences*, **30**, 622–629. 57
- BARLOW, H. (1985). Cerebral cortex as model builder. *Models of the visual cortex*, 37–46. 119
- BARRETT, A. & VAN ROSSUM, M. (2008). Optimal learning rules for discrete synapses. *PLoS Comput Biol*, **4**, e1000230. 23

REFERENCES

- BEAR, M., CONNORS, B. & PARADISO, M. (2007). *Neuroscience: Exploring the brain*. Lippincott Williams & Wilkins. 2
- BIENENSTOCK, E. (1995). A model of neocortex. *Network: Computation in Neural Systems*, **6**, 179–224. 111, 113
- BLISS, T. & LØMO, T. (1973). Long-lasting potentiation of synaptic transmission in the dentate area of the anaesthetized rabbit following stimulation of the perforant path. *The Journal of Physiology*, **232**, 331. 111, 139
- BLOOM, B. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, **13**, 422–426. 9
- BOGACZ, R., BROWN, M. & GIRAUD-CARRIER, C. (1999). High capacity neural networks for familiarity discrimination. *Proceedings of ICANN'99, Edinburgh*, 773–778. 22, 23
- BOLSHAKOV, V., GOLAN, H., KANDEL, E. & SIEGELBAUM, S. (1997). Recruitment of new sites of synaptic transmission during the cAMP-dependent late phase of LTP at CA3-CA1 synapses in the hippocampus. *Neuron*, **19**, 635–651. 75
- BOSE, R. & RAY-CHAUDHURI, D. (1960). On a class of error correcting binary group codes. *Information and control*, **3**, 68–79. 85
- BRANCO, T., CLARK, B.A. & HÄUSSER, M. (2010). Dendritic Discrimination of Temporal Input Sequences in Cortical Neurons. *Science*, **329**, 1671–1675. 116
- BRODER, A. & MITZENMACHER, M. (2004). Network applications of Bloom filters: A survey. *Internet Mathematics*, **1**, 485–509. 9, 24
- CHECHIK, G., MEILIJON, I. & RUPPIN, E. (2001). Effective neuronal learning with ineffective Hebbian learning rules. *Neural Computation*, **13**, 817–840. 74
- CHKLOVSKII, D., SCHIKORSKI, T. & STEVENS, C. (2002). Wiring optimization in cortical circuits. *Neuron*, **34**, 341–347. 107
- DAYAN, P. & WILLSHAW, D. (1991). Optimising synaptic learning rules in linear associative memories. *Biological cybernetics*, **65**, 253–265. 74

REFERENCES

- FAISAL, A. & LAUGHLIN, S. (2007). Stochastic simulations on the reliability of action potential propagation in thin axons. *PLoS Comput Biol*, **3**, 783–795. 140
- FAN, L., CAO, P., ALMEIDA, J. & BRODER, A. (2000). Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, **8**, 293. 79
- FUSI, S. & ABBOTT, L. (2007). Limits on the memory storage capacity of bounded synapses. *Nature Neuroscience*, **10**, 485–493. 55
- GERSTNER, W., RITZ, R. & VAN HEMMEN, J. (1993). Why spikes? Hebbian learning and retrieval of time-resolved excitation patterns. *Biological Cybernetics*, **69**, 503–515. 111
- GERSTNER, W., KEMPTER, R., VAN HEMMEN, J. & WAGNER, H. (1996). A neuronal learning rule for sub-millisecond temporal coding. *Nature*, **383**, 76–78. 137
- GOLDING, N., STAFF, N. & SPRUSTON, N. (2002). Dendritic spikes as a mechanism for cooperative long-term potentiation. *Nature*, **418**, 326–331. 116
- GOODRICH, M. & TAMASSIA, R. (2009). *Data structures and algorithms in Java*. Wiley-India. 6
- GRAHAM, B. & WILLSHAW, D. (1995). Improving recall from an associative memory. *Biological Cybernetics*, **72**, 337–346, 10.1007/BF00202789. 20
- GRAHAM, B. & WILLSHAW, D. (1996). Information efficiency of the associative net at arbitrary coding rates. *Artificial Neural Networks–ICANN 96*, 35–40. 20
- GRAHAM, B. & WILLSHAW, D. (1997). Capacity and information efficiency of the associative net. *Network: Computation in Neural Systems*, **8**, 35–54. 20
- GREVE, A., STERRATT, D., DONALDSON, D., WILLSHAW, D. & VAN ROSSUM, M. (2009). Optimal learning rules for familiarity detection. *Biological Cybernetics*, **100**, 11–19, 10.1007/s00422-008-0275-4. 22, 23
- HENNIG, P. (2011). *Approximate Inference in Graphical Models*. Ph.D. thesis, Cambridge University. 35

REFERENCES

- HENSON, R. (1993). *Short-term associative memories*. Master's thesis, University of Edinburgh. Department of Artificial Intelligence. 56
- HERRMANN, M., HERTZ, J. & PRÜGEL-BENNETT, A. (1995). Analysis of synfire chains. *Network: Computation in neural systems*, **6**, 403–414. 113
- HERTZ, J., KROGH, A. & PALMER, R. (1991). *Introduction to the theory of neural computation*. Westview press. 81
- HINTON, G., MCCLELLAND, J. & RUMELHART, D. (1984). *Distributed representations*. Carnegie Mellon University, Computer Science Dept. 2
- HINTON, G.E. & VAN CAMP, D. (1993). Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the sixth annual conference on Computational learning theory*, COLT '93, 5–13, ACM, New York, NY, USA. 1, 6
- HOCQUENGHEM, A. (1959). Codes correcteurs d'erreurs. *Chiffres*, **2**, 147–56. 85
- HOPFIELD, J.J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, **79**, 2554. 58, 63, 64, 66
- HOPFIELD, J.J. (1995). Pattern recognition computation using action potential timing for stimulus representation. *Nature*, **376**, 33–36. 111
- HOPPENSTEADT, F. & IZHIKEVICH, E. (2003). In M. Arbib, ed., *The handbook of brain theory and neural networks*, chap. Canonical Neural Models, The MIT Press. 114
- IKEGAYA, Y., AARON, G., COSSART, R., ARONOV, D., LAMPL, I., FERSTER, D. & YUSTE, R. (2004). Synfire chains and cortical songs: temporal modules of cortical activity. *Science*, **304**, 559. 113
- IZHIKEVICH, E. (2004). Simple model of spiking neurons. *Neural Networks, IEEE Transactions on*, **14**, 1569–1572. 114

REFERENCES

- IZHIKEVICH, E. (2006). Polychronization: Computation with spikes. *Neural Computation*, **18**, 245–282. 111, 113, 114, 120
- IZHIKEVICH, E.M. & EDELMAN, G.M. (2008). Large-scale model of mammalian thalamocortical systems. *Proceedings of the National Academy of Sciences*, **105**, 3593–3598. 114
- JI, D. & WILSON, M. (2006). Coordinated memory replay in the visual cortex and hippocampus during sleep. *Nature Neuroscience*, **10**, 100–107. 128
- KANERVA, P. (1988). *Sparse distributed memory*. The MIT Press. 42, 49, 57, 58
- KIRSCH, A. & MITZENMACHER, M. (2005). Building a better bloom filter. Tech. Rep. TR-02-05, Harvard University. 12
- KIRSCH, A. & MITZENMACHER, M. (2006). Distance-sensitive bloom filters. In *Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments and the Third Workshop on Analytic Algorithmics and Combinatorics*. 42
- KIRSCH, A. & MITZENMACHER, M. (2008). Less hashing, same performance: Building a better Bloom filter. *Random Structures and Algorithms*, **33**, 187–218. 24
- KNOBLAUCH, A. (2010a). Optimal synaptic learning in non-linear associative memory. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, 1–7, IEEE. 75
- KNOBLAUCH, A. (2010b). Zip nets: Efficient associative computation with binary synapses. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, 1–8, IEEE. 75
- KNOBLAUCH, A., PALM, G. & SOMMER, F. (2010). Memory capacities for synaptic and structural plasticity. *Neural Computation*, **22**, 289–341. 43, 144
- KÖNIG, P., ENGEL, A. & SINGER, W. (1996). Integrator or coincidence detector? The role of the cortical neuron revisited. *Trends in Neurosciences*, **19**, 130–137. 116
- KSCHISCHANG, F., FREY, B. & LOELIGER, H. (2001). Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory*, **47**, 498–519. 35

REFERENCES

- LAZAR, A., PIPA, G. & TRIESCH, J. (2009). SORN: a self-organizing recurrent neural network. *Frontiers in Computational Neuroscience*, **3**, 116
- LEE, K., CHUNG, K., CHUNG, J. & COGGESHALL, R. (1986). Correlation of cell body size, axon size, and signal conduction velocity for individually labelled dorsal root ganglion cells in the cat. *The Journal of Comparative Neurology*, **243**, 335–346. 139
- LENGYEL, M., KWAG, J., PAULSEN, O. & DAYAN, P. (2005). Matching storage and recall: hippocampal spike timing-dependent plasticity and phase response curves. *Nature Neuroscience*, **8**, 1677–1683. 68, 74
- LIU, G. (2004). Local structural balance and functional interaction of excitatory and inhibitory synapses in hippocampal dendrites. *Nature Neuroscience*, **7**, 373–379. 116
- LONDON, M. & HÄUSSER, M. (2005). Dendritic computation. *Annual Review of Neuroscience*, **28**, 503–532. 116
- LUBENOV, E.V. & SIAPAS, A.G. (2008). Decoupling through synchrony in neuronal circuits with propagation delays. *Neuron*, **58**, 118 – 131. 116
- MACKAY, D. & NEAL, R. (1995). Good codes based on very sparse matrices. *Lecture Notes in Computer Science*, **1025**, 100–111. 92
- MACKAY, D. & NEAL, R. (1997). Near Shannon limit performance of low density parity check codes. *Electronics letters*, **33**, 457–458. 86
- MACKAY, D.J.C. (1991). Maximum entropy connections: neural networks. In W.T. Grandy & L. Schick, eds., *Maximum Entropy and Bayesian Methods*, Laramie, Kluwer, Dordrecht. 66, 69, 70
- MACKAY, D.J.C. (2003). *Information Theory, Inference, and Learning Algorithms*. Cambridge University Press, available from <http://www.inference.phy.cam.ac.uk/mackay/itila/>. 32, 36, 43, 72, 85, 86

REFERENCES

- MARKRAM, H. (2006). The blue brain project. *Nature Reviews Neuroscience*, **7**, 153–160. 2
- MARKRAM, H., LUBKE, J., FROTSCHER, M. & SAKMANN, B. (1997). Regulation of synaptic efficacy by coincidence of postsynaptic APs and EPSPs. *Science*, **275**, 213. 116
- MÉZARD, M., NADAL, J. & TOULOUSE, G. (1986). Solvable models of working memories. *Journal de physique*, **47**, 1457–1462. 51
- MILLER, R. (1987). Representation of brief temporal patterns, Hebbian synapses, and the left-hemisphere dominance for phoneme recognition. *Psychobiology*, **15**, 241–247. 139
- MITZENMACHER, M. & UPFAL, E. (2005). *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press. 6, 10, 12, 101
- MOUNTCASTLE, V. (1997). The columnar organization of the neocortex. *Brain*, **120**, 701. 1
- MURPHY, K., WEISS, Y. & JORDAN, M. (1999). Loopy belief propagation for approximate inference: An empirical study. In *Proceedings of Uncertainty in AI*, 467–475. 36
- OENNING, T. & MOON, J. (2001). A low-density generator matrix interpretation of parallel concatenated single bit parity codes. *IEEE Transactions on Magnetics*, **37**, 737. 86
- ORAM, M., WIENER, M., LESTIENNE, R. & RICHMOND, B. (1999). Stochastic nature of precisely timed spike patterns in visual system neuronal responses. *Journal of Neurophysiology*, **81**, 3021. 113
- PALM, G. & SOMMER, F. (1992). Information capacity in recurrent McCulloch-Pitts networks with sparsely coded memory states. *Network: Computation in Neural Systems*, **3**, 177–186. 48
- PASCUAL-LEONE, A., AMEDI, A., FREGNI, F. & MERABET, L. (2005). The Plastic Human Brain Cortex. *Neuroscience*, **28**, 377. 1

REFERENCES

- PEARL, J. (1988). *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann. 36
- PLATE, T. (2000). Randomly connected sigma-pi neurons can form associator networks. *Network: Computation in Neural Systems*, **11**, 9, 13, 19, 21
- PLATE, T. (2003). Distributed representations. In L. Nadel & C.R. (Firm), eds., *Encyclopedia of cognitive science*, Nature Publishing Group. 6
- POLSKY, A., MEL, B. & SCHILLER, J. (2004). Computational subunits in thin dendrites of pyramidal cells. *Nature Neuroscience*, **7**, 621–7. 116
- RAKIC, P. (1995). A small step for the cell, a giant leap for mankind: a hypothesis of neocortical expansion during evolution. *Trends in Neurosciences*, **18**, 383 – 388. 1
- RICHARDSON, T. & URBANKE, R. (2001). The capacity of low-density parity-check codes under message-passing decoding. *IEEE Transactions on Information Theory*, **47**, 599–618. 4, 89, 147
- RICHARDSON, T. & URBANKE, R. (2002). Efficient encoding of low-density parity-check codes. *IEEE Transactions on Information Theory*, **47**, 638–656. 86
- RICHARDSON, T. & URBANKE, R. (2008). *Modern coding theory*. Cambridge University Press. 43, 83, 86
- RISSANEN, J. (1986). Stochastic complexity and statistical inference. *Analysis and Optimization of Systems*, 391–407. 1
- ROLSTON, J., WAGENAAR, D. & POTTER, S. (2007). Precisely timed spatiotemporal patterns of neural activity in dissociated cortical cultures. *Neuroscience*, **148**, 294 – 303. 116
- ROMANI, S., AMIT, D.J. & AMIT, Y. (2008). Optimizing one-shot learning with binary synapses. *Neural Computation*, **20**, 1928–1950. 57
- SALAKHUTDINOV, R. & HINTON, G. (2009). Semantic hashing. *International Journal of Approximate Reasoning*, **50**, 969–978. 6

REFERENCES

- SANDBERG, A., LANSNER, A., PETERSSON, K. & EKEBERG, O. (2000). A palimpsest memory based on an incremental Bayesian learning rule. *Neurocomputing*, **32**, 987–994. 49
- SEJNOWSKI, T. (1977). Statistical constraints on synaptic plasticity. *Journal of Theoretical Biology*, **69**, 385. 74
- SHANNON, C. (2001). A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, **5**, 3–55. 84, 85
- SIMONCELLI, E., PANINSKI, L., PILLOW, J. & SCHWARTZ, O. (2004). Characterization of neural responses with stochastic stimuli. *The Cognitive Neurosciences*, 327–338. 112, 116, 117
- SOFTKY, W. (1994). Sub-millisecond coincidence detection in active dendritic trees. *Neuroscience*, **58**, 13–41. 116
- SOMMER, F. & DAYAN, P. (1998). Bayesian retrieval in associative memories with storage errors. *IEEE Transactions on Neural Networks*, **9**, 705–713. 4, 20, 38, 143
- STERRATT, D. & WILLSHAW, D. (2008). Inhomogeneities in heteroassociative memories with linear learning rules. *Neural Computation*, **20**, 311–344. 51, 74
- STORKEY, A. (1998). Palimpsest memories: a new high-capacity forgetful learning rule for Hopfield networks. *Submitted to Physical Review E*. 49
- SWINDALE, N. (1990). Is the cerebral cortex modular? *Trends in Neurosciences*, **13**, 487–492. 107
- VAN LINT, J. (1999). *Introduction to coding theory*. Springer Verlag. 85
- VERTES, P. (2010). *Computation in Spiking Neural Networks*. Ph.D. thesis, Cambridge University. 146
- VERTES, P. & DUKE, T. (2010). Effect of network topology on neuronal encoding based on spatiotemporal patterns of spikes. *HFSP*, **4**, 153–163. 111, 114
- WATT, A. & DESAI, N. (2010). Homeostatic plasticity and STDP: keeping a neuron’s cool in a fluctuating world. *Front. Syn. Neurosci*, **2**, 1–16. 116

REFERENCES

- WEISS, Y. (2000). Correctness of local probability propagation in graphical models with loops. *Neural Computation*, **12**, 1–41. 36
- WERMTER, S., ELSHAW, M., AUSTIN, J. & WILLSHAW, D. (2001). Towards novel neuroscience-inspired computing. *Emergent Neural Computational Architectures Based on Neuroscience*, 1–19. 1
- WHITE, J., SOUTHGATE, E., THOMSON, J. & BRENNER, S. (1986). The structure of the nervous system of the nematode *Caenorhabditis elegans*. *Philosophical Transactions of the Royal Society of London. B, Biological Sciences*, **314**, 1. 146
- WILLS, S. (2004). *Computation with spiking neurons*. Ph.D. thesis, Cambridge University. 109, 111, 113, 115, 122, 126, 139
- WILLSHAW, D. (1971). *Models of Distributed Associative Memory*. Ph.D. thesis, Edinburgh University. 49, 51, 56
- WILLSHAW, D., BUNEMAN, O. & LONGUET-HIGGINS, H. (1969). Non-holographic associative memory. *Nature*, **222**, 960–962. 9, 19
- WILSON, R. (2009). Parallel Hopfield networks. *Neural Computation*, **21**, 831–850. 115, 139