Noname manuscript No.
(will be inserted by the editor)

Efficient and Robust Associative Memory from a Generalized Bloom Filter

Philip Sterne

Received: date / Accepted: date

Abstract We develop a variant of a Bloom filter that is robust to hardware failure and show how it can be used as an efficient associative memory. We define a measure of the information recall and show that our new associative memory is able to recall more than twice as much information as a Hopfield network. The extra efficiency of our associative memory is all the more remarkable as it uses only bits while the Hopfield network uses integers.

Keywords Associative Memory · Randomized Storage · Bloom Filter

1 Introduction

A *Bloom filter* is a data structure that provides a probabilistic test of whether a new item belongs to a set (Broder and Mitzenmacher, 2004). In this paper we generalize the Bloom filter in such a way that we can perform inference over the set of stored items. This inference allows the filter to serve as an efficient and robust associative memory.

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. Our network however, uses principled statistical inference similar in spirit to (Sommer and Dayan, 1998), but for a much more general architecture.

The approach described here 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 the results of simple random functions of the input and performing distributed statistical inference on the stored values. Each individual part of our Bloom filter works in parallel, independently of each other. Since the parts are independent, the inference is able to continue even if some of the parts fail.

P. Sterne
Cavendish Laboratory
Cambridge University
Present address:
Frankfurt Institute for Advanced Studies
Johann Wolfgang Goethe University
Ruth-Moufang-Str. 1
60438 Frankfurt am Main
Tel.: +49-(0)69-798-47652
Fax: +49-(0)69-798-47611
E-mail: sterne@fias.uni-frankfurt.de

2 Neurally-inspired Bloom filter

A standard Bloom filter is a representation of a set of items. The algorithm has a small (but controllable) probability of a false positive. This means there is a small probability of the Bloom filter reporting that an item has been stored when in fact it has 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.

In the next section we briefly describe a standard Bloom filter before moving on to the neurally-inspired version. A good introduction to Bloom filters is given in Mitzenmacher and Upfal (2005).

2.1 The standard Bloom filter

A standard Bloom filter consists of a bit vector z, and K hash functions $\{h_1, \ldots h_K\}$. Each hash function is a map from the object space onto indices in z. To represent the set we initialise all elements of z to zero. For every object x that

is added to the set we calculate $\{h_1(x), \ldots, h_K(x)\}$, using the results as indices, and set the bit at each index. If a bit has already been set, then it remains on.

To test whether or not $\hat{x}$ is an element of the set we calculate $h_1(\hat{x}), \ldots, h_K(\hat{x})$; if any of the locations $h_k(x)$ in z contains 0 then we can be certain the item is not in the set. If all indices are 1, then $\hat{x}$ is probably part of the set, although it is possible that it is not and all K bits were turned on by storing other elements.

The calculations for a Bloom filter can also be thought of as occurring at each individual bit in z. The $z_m{}^{\text{th}}$ bit calculates all the hash functions and if any of the hash functions are equal to m then z_m is set. To literally wire a Bloom filter in this way would be wasteful, as the same computation is being repeated for each bit. However this view suggests a generalization to the Bloom filter. Rather than calculate a complicated hash function for each bit, a neurally-inspired Bloom filter calculates a random boolean function of a subset of the bits in x.

2.2 Neurally-inspired modifications

A Bloom filter can be turned into a neurally-inspired Bloom filter (as shown in figure 1) by replacing the K hash functions with a simple boolean function for each bit in z. 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, but M.

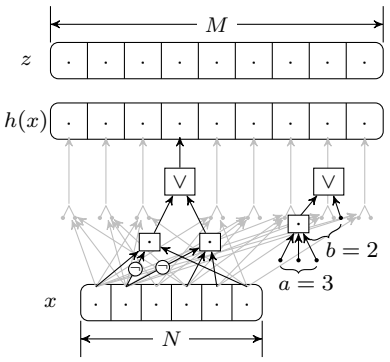
To initialise the network we set the storage z to zero. We store information by presenting the network with the input and whichever boolean functions return true, the associated storage bit z_m is turned on. To query whether an $\hat{x}$ has been stored, we evaluate $h(\hat{x})$ and if any of the boolean functions return true, but the corresponding bit in z is false 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.

For our boolean function, we 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[1](Plate, 2000):

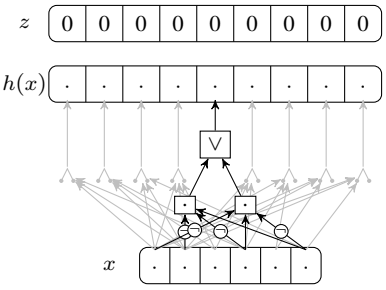
$$h_m(x) = x_1 \cdot (\neg x_6) \cdot x_3 \vee (\neg x_3) \cdot x_9 \cdot (\neg x_2) \vee (\neg x_5) \cdot x_6 \cdot x_4. \tag{1}$$

The variables that are "AND"ed together are chosen at random, and varied for each bit of storage. These functions have the advantage that they are simple to evaluate, and they

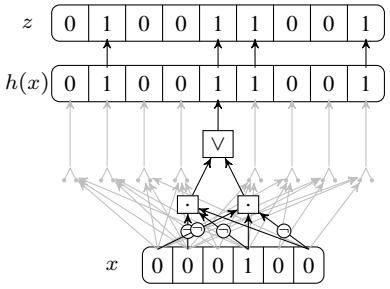
[1] We dislike the sigma-pi terminology and find the term "sum-product" function to be much more descriptive. However this terminology is unfortunately confusing as belief propagation is also known as the sum-product algorithm.



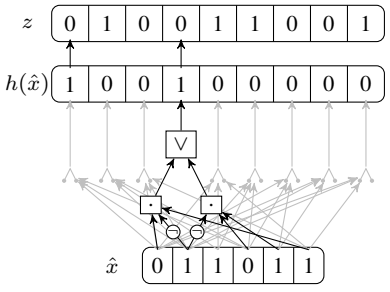
(a) Each bit has a function which "AND"s many variables and "OR"s the result.



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



(c) Adding an item.



(d) Testing for membership.

Fig. 1 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.

have a tunable probability of returning true in the absence of knowledge about x.

Given the number R of items to store, we can find the form of the boolean functions that minimize the false positive probability rate. 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:

$$p^* = 1 - (1-p)^R. \tag{2}$$

We can estimate the false positive probability, by noting that there are $M(1-p)^R$ zeros and the probability of all of these returning false is:

$$\begin{aligned} \Pr(\text{failure}) &= (1-p)^{M(1-p)^R} \\ &\approx e^{-Mp(1-p)^R}, \end{aligned} \tag{3}$$

which has a minimum when:

$$p = \frac{1}{R+1} \tag{4}$$

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 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:

$$p = 1 - \left(1 - 2^{-a}\right)^b, \tag{5}$$

where a is the number of "AND"s and b is the number of "OR"s. For large R and equation 4, this can be approximated by the relation:

$$b = \frac{2^a}{R+1}. \tag{6}$$

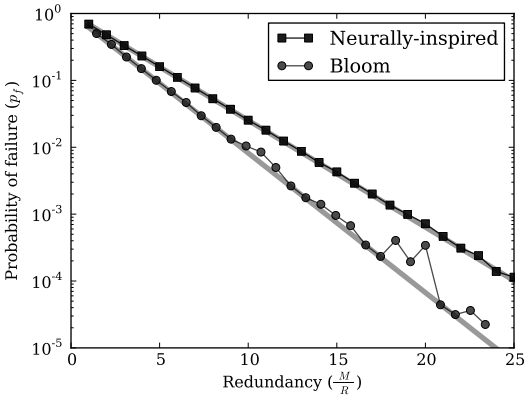
Note that the function does not depend on M. The neurally-inspired Bloom filter scales with any amount of storage, while optimal performance in a traditional Bloom filter requires the number of hash functions be changed. This optimality with regards to hardware failure makes the neurally-inspired Bloom filter robust. 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:

$$M = R\frac{-\ln p_f}{\ln^2 2}, \tag{7}$$

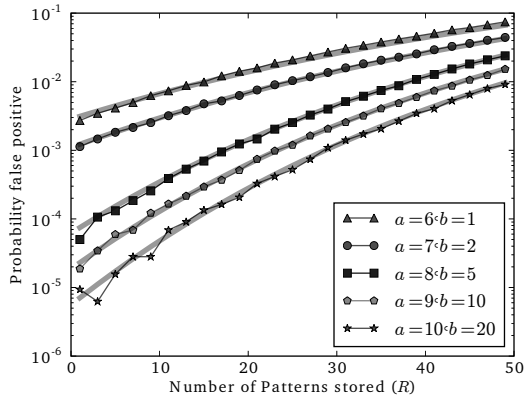
while a neurally-inspired Bloom filter would require a storage size:

$$M = (R+1)(-\ln p_f)e. \tag{8}$$

For an equal probability of bit failure the neurally-inspired version requires 31% ($= e\ln^2 2$) more storage. This is the price for choosing bits in parallel, independently of all others. If we could 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.



(a) The false probability of the Bloom filter and its neurally-inspired equivalent fall off exponentially as more storage is added.



(b) The false positive probability for an input which differs in only one bit from a stored pattern. Increasing the number of "AND"s decreases the probability of error, but requires exponentially more "OR"s.

Fig. 2 The probability of error for a neurally-inspired Bloom filter for completely random binary vectors and vectors that only differ in a single bit. The grey lines give the expected theoretical values (as derived from equation 8 and 9 respectively), which are closely matched by the experiments.

In figure 2(a) we compare the theoretical performance with the empirical perfomance of both the Bloom filter and the neurally-inspired Bloom filter. The dotted lines represent the theoretical performance of each and the empirical performance of both are very close to their theoretical performance. The empirical performance was obtained by averaging many runs together, using different boolean hash functions. In general we found that practically all neurally-

inspired Bloom filters were well-behaved and the performance was close to the mean.

To obtain these plots we stored $R = 100$ random patterns of $N = 32$ bits. We let M range from $M = 100$ to $M = 2500$. Note that even with the maximum storage we tested we still used less bits than naively storing the patterns would use ($NR = 3200$). The h_m functions were constucted by "AND"ing $a = 10$ distinct variables together and "OR"ing $b = 10$ such terms together. The many hash functions of a traditional Bloom filter were constructed using a linear composition of two hash functions as suggested by Kirsch and Mitzenmacher (2008), the two hash functions themselves were linear multiplicative hash functions taken from Knuth (1973).

2.3 Sensitivity to a bit flip

Since the boolean hash functions only use a small number of the bits in x we might expect that the probability of error for an unstored x which is very similar to a stored pattern will be much higher 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 in h_m the probability that it contains the flipped bit is $\frac{a}{N}$. Therefore the probability of a hash function returning true given a single bit flip is $p \cdot \frac{a}{N}$. 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 are turned on:

$$\begin{aligned} \Pr(\text{failure} \,|\, \text{single bit flip}) &= \left[1 - \frac{ap}{N}\right]^{M(1-p)^R} \\ &\approx e^{-Mp(1-p)^R \cdot \frac{a}{N}} \\ &= \left[\Pr(\text{failure})\right]^{\frac{a}{N}} . \end{aligned} \tag{9}$$

(This approximation is only accurate when the total number of possible boolean hash functions is much larger than M, the number actually used.)

Equation 9 also shows that as more bits are "AND"ed together the performance for neighboring binary vectors becomes more independent, and in the case $a = N$ the performance is completely independent. Practically though only a small number of bits can be "AND"ed together as the number of "OR"s required increases exponentially (equation 6). Our results in figure 2(b) show that the approximations are accurate. This plot set $M = 2000$, $N = 32$, stored $R = 50$ patterns and we let a range from 5 to 10, while keeping the probability of $h(x)$ returning true as close to optimal as possible. It is clear that increasing a decreases the probability of a false positive from a single bit flip.

A neurally-inspired Bloom filter underperforms a standard Bloom filter as it requires more storage and has a higher rate of errors for items that are similar to already-stored items. While such negative results might cause one to discard the neurally-inspired Bloom filter, we show that there is still an area where it outperforms. The neurally-inspired Bloom filter's simple hash functions allow us to perform inference over all possible x vectors. This allows us to create an associative memory with high capacity, which we show in the next section.

2.4 Related work

Particularly relevant work to this article 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. 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 8 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:

$$R = \frac{N(N-1)}{2 \times 13} \tag{10}$$

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. Since our representation of a set is more efficient than a Hopfield network we might suspect that we can create a more efficient associative memory as well.

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 all zeros. When we store a pattern we calculate the indexes into z and set those entries to 1. So far 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:

$$h_k(x) = 16x_{22} + 8x_{43} + 4x_6 + 2x_{65} + x_{75}. \tag{11}$$

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 11 into the following set of sigma-pi functions:

$$\begin{aligned} h_0(x) &= (\neg x_{22})(\neg x_{43})(\neg x_6)(\neg x_{65})(\neg x_{75}) \\ h_1(x) &= (\neg x_{22})(\neg x_{43})(\neg x_6)(\neg x_{65})(x_{75}) \\ h_2(x) &= (\neg x_{22})(\neg x_{43})(\neg x_6)(x_{65})(\neg x_{75}) \\ &\cdots \\ h_{31}(x) &= (x_{22})(x_{43})(x_6)(x_{65})(x_{75}) \end{aligned} \tag{12}$$

However we have a different aim, in that we want to be sensitive to different patterns, even if they are similar. In the next section 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.

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 3). During recall time just an x is presented and the task is to use the synaptic connectivity to estimate what y should be. While many recall rules have been proposed (Graham and Willshaw, 1996, 1995, 1997) the most relevant performs proper statistical inference (Sommer and 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:

$$R = \frac{N}{\log_2 N} \tag{13}$$

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 yy' is:

$$\Pr(\text{failure}) = 2^{-(\log_2 \mathrm{N})^2} = \mathrm{N}^{-\log_2 \mathrm{N}}. \tag{14}$$

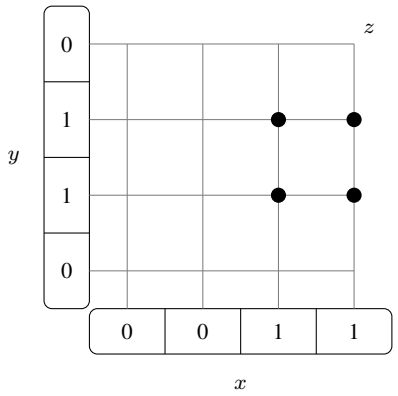


Fig. 3 The Willshaw associative network stores pairs of patterns $\{(x, y)\}$. To store a pattern z_{ij} is updated according to the following rule: $z_{ij} = z_{ij} \vee x_i y_j$. The Willshaw network is designed to store sparse binary patterns. Given a correct x we can then use z to recall the associated y.

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 as the pattern sizes increase:

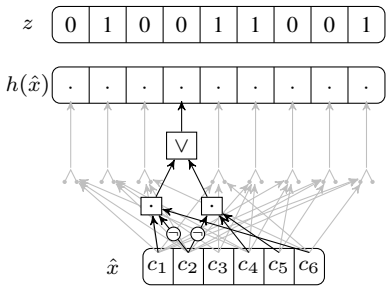
$$R \approx \frac{-N}{\log_2 N} \cdot \log\left(1 - p_f^{\left(\frac{1}{\log_2 N}\right)^2}\right). \tag{15}$$

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).

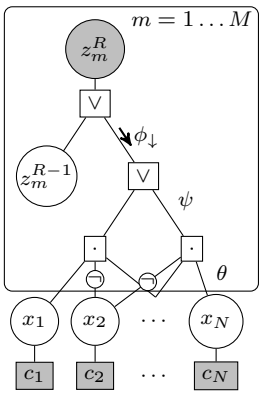
3 Neurally-inspired Binary associative memory

An associative memory is similar to a Bloom filter as one can store many patterns in memory, but rather than querying for an exact match, an approximate pattern is provided and the task is to provide the closest stored item.

The neurally-inspired associative memory is constructed directly from the neurally-inspired Bloom filter of the preceding section. Naively one could iterate through all possible binary vectors and use the Bloom filter to rule out most of them. We could use the prior to assign preferences to the remaining vectors. To predict a single bit in x the proper 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, however there is an message-passing algorithm which can use the simple, local boolean hash functions and is able to approximate just such an algorithm.



(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 $\Pr(x_n \mid c, z)$.



(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.

Fig. 4 The associative memory task and the factor graph that achieves that task.

The most relevant literature for this section is Kirsch and 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.

3.1 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: $\Pr(\hat{x}_n) = c_n$. To decode the noisy redundant vector we use loopy belief propagation (MacKay, 2003), (Bishop, 2006).

If we cared about recalling the entire x correctly, then we would want to find the most likely x. However in this paper we tolerate a small number of bit errors in x. To minimize the 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 z and the noisy cue c is given by Bayes' rule:

$$\Pr(x \mid c, z) = \frac{\Pr(x \mid c) \cdot \Pr(z \mid x)}{\Pr(z)}, \tag{16}$$

where $\Pr(x \mid c)$ represents the prior probability of x given an initial cue. We assume that noise in x is independent and all bits are flipped with identical probability. To make a prediction for an individual bit x_n we need to marginalise out over all other bits $\Pr(x_n \mid c, z) = \sum_{x_{-n}} \Pr(x \mid c, z)$ (where x_{-n} refers to changing all possible values of x except the n^{th} value).

The marginal calculation can be simplified as the distributive law applies to the sum of many products. In general the problem is broken down into message passing along the edges of the graph shown in figure 4(a). In figure 4(b) the equivalent factor graph is shown. For a more detailed explanation of the sum-product algorithm (which is a slightly more general version of belief propagation) see Kschischang et al (2001). Unfortunately the sum-product algorithm is only exact for graphs which form a tree, and the graph considered here is not a tree. However the algorithm can be viewed as an approximation to the real posterior (MacKay, 2003) and empirically gives good performance.

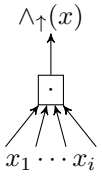
3.2 Calculation of Messages

We use the index m to refer to functions since we have M encoding functions, and use the index n to refer to the variables (although there are more than N variables, e.g the intermediate calculations for each boolean hash function). 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:

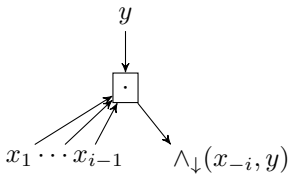
$$\neg \begin{bmatrix} \bar{x}_n \\ x_n \end{bmatrix} = \begin{bmatrix} x_n \\ \bar{x}_n \end{bmatrix}. \tag{17}$$

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:



$$\wedge_\uparrow(x) = \begin{bmatrix} 1 - \prod_i x_i \\ \prod_i x_i \end{bmatrix}. \tag{18}$$

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



$$\wedge_\downarrow(x_{-i}, y) = \begin{bmatrix} \bar{y} \\ \bar{y} + (y - \bar{y}) \left(\prod_{j \neq i} x_j \right) \end{bmatrix} \tag{19}$$

The "OR" message can be derived from the observation that:

$$x_1 \vee x_2 = y \tag{20}$$

is logically equivalent to:

$$(\neg x_1) \wedge (\neg x_2) = \neg y, \tag{21}$$

which gives:

$$\vee_\uparrow(x) = \begin{bmatrix} \prod_i \bar{x}_i \\ 1 - \prod_i \bar{x}_i \end{bmatrix} \tag{22}$$

and

$$\vee_\downarrow(x_{-i}, y) = \begin{bmatrix} y + (\bar{y} - y) \left(\prod_{j \neq i} \bar{x}_j \right) \\ y \end{bmatrix}. \tag{23}$$

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

$$\begin{bmatrix} \bar{\phi}_m \\ \phi_m \end{bmatrix} = \begin{bmatrix} 1 - z_m p^{R-1} \\ z_m \end{bmatrix}, \tag{24}$$

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:

$$\Pr(x_n \mid z, c) \propto c_n \cdot \prod_m \theta_{m \downarrow n} \tag{25}$$

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:

$$\hat{x}_n = \mathbb{1}\left[\Pr(x_n \mid z, c) > 0.5\right]. \tag{26}$$

We now provide a metric of the recall performance that can be used for both probabilistic associative memories and non-probabilistic ones.

3.3 Performance measure

To evaluate how well the memory is performing we use results from the binary symmetric channel to measure the information in the recall and define the performance as:

$$s(p) = N \cdot (1 - H_2(p)), \tag{27}$$

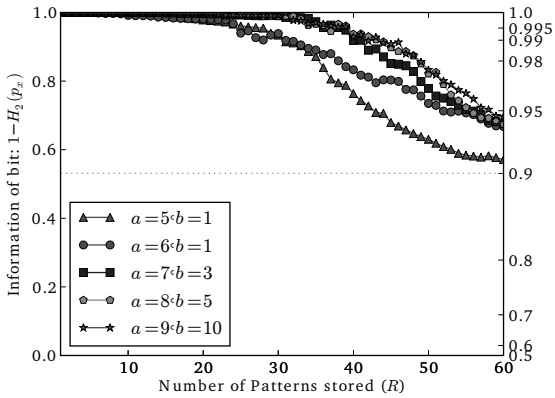
where H_2 is the standard binary entropy function:

$$H_2(p) = -p \log_2 p - (1 - p) \log_2 (1 - p). \tag{28}$$

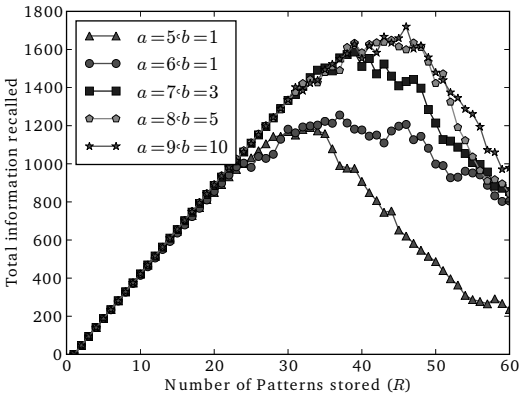
It is also important to take into account the information in the cue. Generally networks that suffer catastrophic failure simply define capacity as the average number of patterns that can be stored before failure occurs. Other networks look at the number of patterns that can be stored before the error rate is worse than a threshold. However this threshold is arbitrary and the performance also depends on the noise present in the cue. Most associative memory work avoids this dependence by using noiseless cues. We take the view that an 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:

$$\begin{aligned} I(p_c) &= (\text{Recalled information}) - (\text{Cue information}) \\ &= s(p_x) - s(p_c) \\ &= N \cdot (H_2(p_c) - H_2(p_x)) \end{aligned} \tag{29}$$

In figure 5(a) we show the information returned from a single recall as more patterns are stored in the network. (On the right hand side of figure 5(a) we show the associated probability of error.) In figure 5(b) we show how much information in total can be recalled by the network. To correctly set the parameters a and b we used some theoretical insight into the performance of the memory.



(a) The recalled information as more patterns are stored, plotted for different numbers of terms "AND"ed together in the boolean hash functions. The dashed line represents simply returning the input cue ($p_c = 0.1$).



(b) The total amount of information recallable by the network is plotted as more patterns are stored.

Fig. 5 The performance of the associative memory as more patterns are stored. For these plots $p_c = 0.1$, $N = 100$ and $M = 4950$.

3.4 Theoretical prediction of the bit error rate

The probability of error during recall in the neurally-inspired associative memory is affected by several possible errors (see figure 6 for a diagram of all the conditions). From equation 3 we know the probability of error of a Bloom filter for a randomly chosen binary vector as part of the set. The uncertainty in the cue means that we expect to see the answer with approximately Np_c bits flipped and there are approximately $\binom{N}{p_c N}$ such binary vectors. The probability of a false positive in the entire set of possible vectors is:

$$\xi_1 : \Pr(\text{False positive}) = 1 - \left(1 - e^{-Mp(1-p)^R}\right)^{2^{NH_2(p_c)}}, \tag{30}$$

where we used the approximation $NH_2(p_c) \approx \log_2 \binom{N}{p_c N}$. There is also an error if another pattern falls within the expected radius:

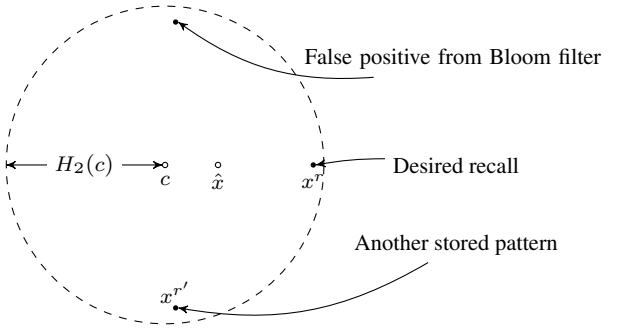


Fig. 6 The uncertainty in the input cue determines a ball of possible binary vectors to recall. When N is large almost all of the mass of this high-dimensional ball will be at the edges. The Bloom filter is able to rule out almost all of the binary vectors. In this diagram we show all possibilities that 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.

$$\xi_2 : \Pr(\text{Other pattern}) = 1 - \left(1 - (R-1)2^{-N}\right)^{2^{NH_2(p_c)}}. \tag{31}$$

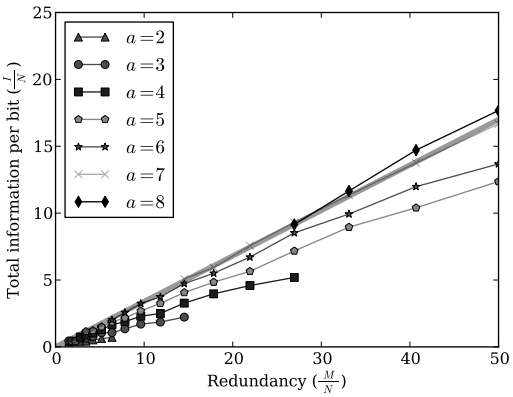
The final error condition (ξ_3) occurs in the region around the desired pattern. A false positive is more likely for binary vectors that differ only by a single bit from a known pattern. This probability of error was derived in equation 9. If the initial cue differs from the stored pattern in the bit that is also accepted as a false positive, then it is 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:

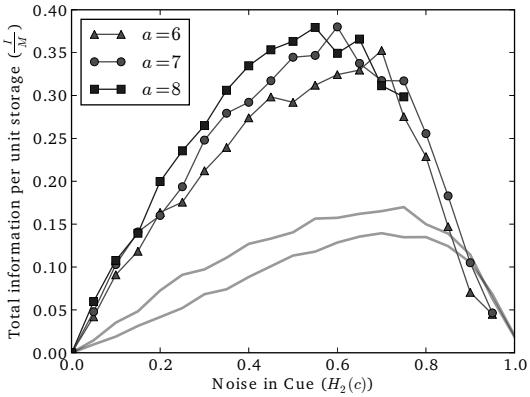
$$\Pr(\text{final bit error}) = \Pr(p_c) \cdot \Pr(\xi_1 \vee \xi_2 \vee \xi_3). \tag{32}$$

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. This simplification gives a poor prediction of the error rate, but still predicts accurately the transition from reliable to unreliable behaviour.

Another reason the error rate is poorly predicted is due to the assumption that any error condition 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



(a) The capacity of the network is linear in the redundancy. The grey line gives a fit to the best capacity. The network efficiency is the the slope of the line.



(b) The information efficiency as a function of the cue noise. The grey lines give equivalent performance of a Hopfield network under different recall strategies.

Fig. 7 The network performance as the redundancy is scaled and the level of noise in the cue is changed.

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. At this transition the associative memory is providing the most information, so the approximation makes acceptable predictions for the best parameter settings. We call the optimal performance the capacity of the associative memory and examine it next.

3.5 Capacity of the neurally-inspired associative memory

In figure 7(a) we show that the capacity of the network scales linearly as M is increased. The rate at which the capacity scales is the efficiency of the network. In figure 7(b) we can see that our definition of the capacity depends on the reliability of the cue. At the optimal cue reliability the neurally-inspired associative memory has an efficiency of 0.37 bits per bit of storage. For comparison the efficiency of a traditional Hopfield network (Hopfield, 1982) is approximately 0.14 bits per unit of storage, which with smart decoding (Lengyel et al, 2005) can be enhanced to 0.17 bits per stored integer (see Sterne (2011) for more details). This is an unfair comparison however as Hopfield networks are able to store integers, rather than a single bit which makes the performance even more impressive.

4 Discussion

This paper 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 requires 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 using probabilistic decoding.

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 simple computational elements that are crude and not directly comparable with biological neurons, we hope that it provides insight into how such memories might be created in the brain.

Acknowledgements I would like to thank David MacKay and members of the Inference group for very productive discussions of the ideas in this paper.

References

Aleksander I, Thomas W, Bowden P (1984) WISARD, a radical step forward in image recognition. Sensor Review 4(3):120–4

Bishop C (2006) Pattern Recognition and Machine Learning. Springer New York:

Bogacz R, Brown M, Giraud-Carrier C (1999) High capacity neural networks for familiarity discrimination. Proceedings of ICANN'99, Edinburgh pp 773–778

Broder A, Mitzenmacher M (2004) Network applications of Bloom filters: A survey. Internet Mathematics 1(4):485–509

Graham B, Willshaw D (1995) Improving recall from an associative memory. Biological Cybernetics 72:337–346, URL http://dx.doi.org/10.1007/BF00202789, 10.1007/BF00202789

Graham B, Willshaw D (1996) Information efficiency of the associative net at arbitrary coding rates. Artificial Neural Networks–ICANN 96 pp 35–40

Graham B, Willshaw D (1997) Capacity and information efficiency of the associative net. Network: Computation in Neural Systems 8(1):35–54

Hopfield JJ (1982) Neural networks and physical systems with emergent collective computational abilities. Proceedings of the National Academy of Sciences 79(8):2554

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

Kirsch A, Mitzenmacher M (2008) Less hashing, same performance: Building a better Bloom filter. Random Structures and Algorithms 33(2):187–218

Knuth D (1973) The art of computer programming. Vol. 3: sorting and searching. Addison Wesley

Kschischang F, Frey B, Loeliger H (2001) Factor graphs and the sum-product algorithm. IEEE Transactions on Information Theory 47(2):498–519

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(12):1677–1683

MacKay DJC (2003) Information Theory, Inference, and Learning Algorithms. Cambridge University Press

Mitzenmacher M, Upfal E (2005) Probability and computing: Randomized algorithms and probabilistic analysis. Cambridge University Press

Plate T (2000) Randomly connected sigma-pi neurons can form associator networks. Network: Computation in Neural Systems 11

Sommer F, Dayan P (1998) Bayesian retrieval in associative memories with storage errors. IEEE Transactions on Neural Networks 9(4):705–713

Sterne P (2011) Distributed Associative Memory. PhD thesis, Cambridge University

Willshaw D, Buneman O, Longuet-Higgins H (1969) Nonholographic associative memory. Nature 222(5197):960–962