Progress Report
December 17 2010 - January 19, 2011

Raj K. Kumar, Amir Hesam Salavati
E-mail: raj.kumar@epfl.ch, hesam.salavati@epfl.ch

Supervisor: Prof. Amin Shokrollahi
E-mail: amin.shokrollahi@epfl.ch

Algorithmics Laboratory (ALGO)
Ecole Polytechnique Federale de Lausanne (EPFL)

January 19, 2011

1 Introduction

In the past month, we considered new models for neural associative memory. More specifically, we investigated non-binary neural networks, i.e. neural networks in which neurons outputs can have *positive integer* values as their output instead of being binary. We can think of such models as a neural networks in which the output of neurons are their firing rate in a short period of time instead of their states. Hence, a given neuron will count the number of spikes fired by its neighbors in a small time window and based on the weighted sum of these firing rates, it decides to fire or not, and if it is going to fire, what would the firing rate be. In fact, continuous valued neurons are more biologically realistic as the input-output relationship of a neuron is a continuous function. Furthermore, neurons have capacitance which results in first integrating the input and then reacting based on that [1]. The only limitations on the output are that it should be a bounded positive number and integer values are preferred.

Non-binary neural networks are specially interesting from a coding point of view since we can use codes in real field to investigate the behavior of such networks. A good example is the use of lattice codes, which we will consider in more details in section 3. The advantage of using lattice codes instead of other coding techniques is that lattice codes operate on the real fields instead of the binary field. Hence, they are more similar to neural networks. We will show that using lattice codes, we will be able to memorize an exponential number of patterns in terms of the number of neurons when noise power is small enough.

In what follows, we start by briefly reviewing a number of papers regarding the storage capacity of Hopfield networks. We will then consider some papers that address non-binary Hopfield networks, including the one by Hopfield himself [1]. In section 3, we will investigate applications of lattice codes in neural networks. Section 4 provides some simulation results about the use of lattice codes as a means of associative memory. Finally, section 5 will conclude this reports and gives a number of ideas fr future works.

2 Literature Review

In the past month, we finished reading the book *Introduction to the the theory of neural computation* by Hertz et al. [2], which is a **must** for anyone working on neuroscience. It covers all the necessary materials from the the basic to the advanced levels.

Then we proceeded by reading some key papers on the storage capacity of Hopfield networks. Among those was the important paper of McEliece et al [3] in which the authors have calculated the storage capacity of the Hopfield networks and showed that if we require all patterns be recalled with high probability, the capacity is proportional to $n/\log(n)$, where n is the number of neurons. Their model is the original Hopfield model in which weights are computed by the outer-product rule, i.e. $w_{ij} = \frac{1}{n}\sum_{\mu=1}^{M} x_i^\mu x_j^\mu$, in which M is the number of patterns and x_i^μ is the i^{th} bit of the pattern μ. Furthermore, all diagonal weights are assumed to be zero. The authors have also considered different variants of the model and showed that the capacity does not change dramatically in those cases. One important case is the one in which we have neurons with three states: ± 1 and erasure where we assign 0 to those neurons which we are not sure about their state. Even this scheme will not cause significant increase in the capacity for large values of n.

Another interesting paper about increasing the storage capacity of Hopfield networks is [4]. The model considered in [4] is the original Hopfield model with a modified weighting rule. The author propose two methods for calculating weights which involves multiplying the $M \times n$ patterns matrix by a *diagonal* matrix followed by multiplying the pseudo-inverse of the patterns matrix. This approach makes sure that the patters are eigenvectors of the weight matrix with positive real eigenvalues.

The authors show that for these two strategies and with *one bit* of error correction, one can memorize up to $n/2$ **random patterns**. This is a significant improvement to the $n/\log(n)$ of [3]. However, it comes with additional computational costs because of the pseudo-inverse operation. Furthermore, except for a special case, the authors do not give a learning rule, similar to the one in original Hopfield approach which calculates the weights gradually from the memorized patterns.

Compared to [4], our work on using Gold sequences is superior as it does not involve any matrix inversion and computes the weights according to the Hopfield rule and we show that the capacity could be increased up to $M = n$, with 6 bits of error correction for $n = 127$. Albeit we considered structures sequences instead of random ones.

2.1 Non-binary Neural Networks

So far, we have only considered binary neural networks in which the state of each neuron is determined by if it has fired or not. However, one can think of the state of each neuron as its short-term firing rate. In other words, we can count the number of spikes a neuron fired in a small time window and take it as the neuron's output. In this model, the output of neuron would be an integer from zero to some maximum value. Such a model gives us more degrees of freedom in designing neural networks for associative memories.

Hopfield himself considered this model shortly after he proposed his famous binary-valued neural networks to implement associative memory. In his paper [1], he shows that continuous valued neurons can also memorize and recall patterns. The model is very similar to the one with binary neurons except for the fact that neurons output is non-binary. The output of the neuron i is a bounded continuous variable denoted by V_i. It is a monotonically increasing function of the input to the neuron, shown by u_i, i.e. $V_i = g_i(u_i)$. g is a function like *sigmoid* or *tanh*. As mentioned before, V_i can be thought of as the short-term average of neuron's firing rate.

The relationship between input to neuron i, i.e. u_i, and the output of other neurons can be modeled as the following equation:

$$C_i \frac{du_i}{dt} = \sum_{j=1}^{n} W_{ij} V_j - \frac{u_i}{R_i} \tag{1}$$

In which W_{ij} is the weight between neurons i and j. C_i and R_i are the capacitance and resistance of neuron i, respectively.

Hopfield shows that if one considers the energy function given in equation (2), then the time evolution of the system based on equation (1) is such that $dE/dt < 0$ and hence the system converges to a minimum.

$$E = -\frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} W_{ij} V_i V_j + \sum_{i=1}^{n} \frac{1}{R_i} \int_0^{V_i} g_i^{-1}(V) dV \tag{2}$$

Hopfield shows that when the nonlinear function $g(.)$ is *steep*, meaning that the transition from its lowest value to its highest is rapid, then the fixed points of the both binary and non-binary models are roughly the same (if the lowest value of g is -1 and highest values is $+1$). As the steepness tends to infinity, they become even more similar which is quite obvious as the shape of the functions becomes more similar to a thresholding unit. However, when the steepness of $g(.)$ is small, the continuous-response model can have *fewer* stable states than the binary model $\underline{\text{with the same weight matrix}}$. Nevertheless, the existing stable states will still correspond to particular stable states of the binary model.

A recent work on non-binary Hopfield networks is [5]. In this paper, we have a network with n *complex-valued* neurons. In the model, neurons can take an integer value from 0 to K. The output of a neuron is determined according this integer by assigning the phase $S_i = e^{i2\pi V_i/K}$, where S_i is the output of neuron i and V_i is its corresponding integer value. This model is mostly suitable for artificial neural networks as it is based on complex values.

The principles of this model is essentially the same as other neural networks: in each iteration, neuron i calculates the weighted input sum of its neighbors, i.e. $h_i \sum_{j=1}^{n} w_{ij} S_j$. The phase of h_i determines S_i, i.e. if $2\pi V/K \leqslant arg(h_i) < 2\pi(V+1)/K$, then $S_i = e^{i2\pi V/K}$. Hence, we have the same problem as we had before: find the *complex* weight matrix W such that the set of given M patterns could be safely memorized and recalled correctly later in presence of noise.

The authors formulate the problem as a linear-satisfiablity problem by considering a set of linear inequalities which ensures the stability of the memorized patterns as well as *one bit of error correction*. The idea is quite interesting: consider an energy function similar to the first term of (2). Find the matrix W such that the memorized patterns are local minimums of such energy function. Furthermore, make sure that there is a radius of attraction around each minimum by considering a neighborhood of distance one around each pattern and design weights such that all patterns with one bit of error converge to the correct pattern with high probability. The authors goes further to eliminate a number of spurious states by optimizing neural thresholds.

The performance of the proposed algorithm in [5] was assessed using simulations over different values of M, n and K. The authors show that when n is large enough, one can memorize up to $M = n$ patterns with $K = 5$ and if K is increased further, the probability of error decreases for $M > n$. However, even with $K = 10$ the probability of error was 79% for $M = 75$ and $n = 50$.

Although the $M = n$ result is quite nice, there are some *minor* drawbacks to the approach proposed in [5]. First of all, operations are based on complex values, which may not be realistic (although we propose an idea in the last section which might resolve this issue). Secondly, just like [4], the authors do not propose any learning rule for the neural network. You have to find the weights using linear optimization techniques. Albeit this task might be accomplished by some neural networks approaches including perceptron learning rules. Compared to our work on Gold sequences, the approach of [5] is less significant as they consider non-binary neurons to achieve $M = n$ with one bit of error correction while we can achieve $M = n$ with binary neurons and 6 bits of error correction for $n = 127$.

Overall, non-binary neurons gives us more degree of freedom in designing neural networks. From a coding theoretical point of view, they will be even better since we can use codes on real fields to come up with neural networks with larger storage capacities. In the next section, we consider application of lattice codes as a means to accomplish this task.

3 Lattice Codes and Neural Associative Memory

In this section, we consider applications of lattice codes in designing neural networks capable of memorizing and recalling patterns in presence of noise. We start by briefly reviewing basics of lattice codes and then propose a weighting scheme based on lattice codes.

3.1 A Brief Introduction to Lattice Codes

A lattice in R^n is a discrete *subgroup* of R^n which spans all real vectors in R^n. In a more formal way, A k dimensional lattice in R^n is defined as the set of all linear combination of a given basis of k linearly independent vectors in R^n **with integer coefficients**. One can find lattice points by multiplying a full rank $k \times n$ generator matrix G with a $1 \times k$ vector of integer coefficients.

In a lattice code, the codewords are intersections of the lattice points with a bounding area (B_n). This area could be hyper-sphere or its sphere. Hence, codewords will be points of a lattice inside a sphere or some region like that. If $\underline{u}$ denotes the $1 \times k$ integer *message* vector, codewords are constructed according to $\underline{x} = \underline{u}G$, where multiplication is done over real field.

Now suppose we have a noisy version of $\underline{x}$, i.e. $\underline{y} = \underline{x} + \underline{e}$, where $\underline{e}$ is the noise vector. If we multiply the pseudo-inverse of G, denoted by G^{-1} for simplicity, we get: $\underline{y}G^{-1} = \underline{x}G^{-1} + \underline{e}G^{-1} = \underline{u} + \underline{e}G^{-1}$. Since $\underline{u}$ is composed on integers, the syndrome of the noisy vector is the fractional part of $\underline{y}G^{-1}$, i.e. $\underline{s} = \underline{e}G^{-1} - round(\underline{e}G^{-1})$. The matrix G^{-1} has the role of the parity check matrix H here.

Similar to LDPC codes, we have Low Density Lattice Codes (LDLCs) where the parity check matrix (G^{-1}) is sparse [6]. In fact, one can visualize the code using the parity check matrix just like the LDPC codes. In LDPC codes, the position of non-zero elements in the parity check matrix is sufficient to specify the matrix. However, in lattice codes we also need the value of the non-zero elements since they are real positive and negative numbers. In this regards, lattice codes look really like a neural weight matrix. Specially in the hetero-association case where we develop an association between the message vector $\underline{u}$ and the code vector $\underline{x}$. LDLCs are even more similar to biological neural networks as the weight matrix is sparse. We will exploit this similarity in the next section to design a neural mechanism of association between these two vectors.

3.2 Lattice Code Model of Hetero-associative Hopfield network

In this section, we will use lattice codes as a model to implement neural mechanisms of associative memory. We consider hetero-association mechanisms and would like to associate a given message vector $\underline{u}$ to its corresponding code vector $\underline{x} = \underline{u}G$, where G is a full rank $k \times n$ matrix with positive integer values. Furthermore, we assume that elements of the message vector $\underline{u}$ can take two integer values: U_{up} and U_{down}. In this way, we ensure that components of $\underline{x}$ are positive integers which can then be modeled as short term firing rate of neurons. In the next section, we will show that the choice $U_{up} = 2$ and $U_{down} = 0$ gives us some good results in terms of error correction.

Figure 1 illustrates the network configuration. $z_1, \dots, z_k$ are message nodes and $y_1, \dots, y_n$ are code nodes. Typically, we have a noisy version of codewords assigned to code nodes and would like to retrieve the corresponding message symbols, i.e. we have $y_i = x_i + e_i$, where e_i is the noise symbol and would like to find the matrix W such that we retrieve the message symbols $z_i = u_i$.

We assume an asymmetric network. More specifically, we assume that the weight matrix is asymmetric such that the $n \times k$ weights from Y nodes

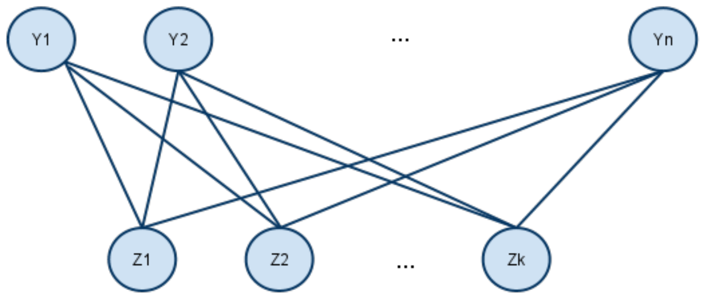

Figure 1: Hetero-associative Hopfield Network

to Z nodes is denoted by W and the $k \times n$ weight matrix from Z nodes to Y nodes is represented by W'. Furthermore, we assume that code nodes (Y nodes) can have an integer value from 0 to an upper bound as their output while message nodes can have only two integer values: *up* and *down*.

The neural update rule is based on the weighted input sum as before. However, message node z_i computes the weighted sum $h_i = \sum_{j=1}^{n} w_{ji} y_j$ and compare it with a threshold θ_i. If $h_i > \theta_i$, then $z_i = up$ and $z_i = down$ otherwise. On the other hand, code node y_j computes the weighted input sum $h'_i = \sum_{i=1}^{k} w'_{ij} z_i$ and $y_j = \max(round(h'_i), h_m ax)$, where $h_m ax$ is the maximum output value a code node can have.

3.2.1 Choice of Weight Matrices

Since we have assumed that we have codewords of a lattice code as the patterns, we have $\underline{x} = \underline{u}G$. Hence, $\underline{y} = \underline{u}G + \underline{e}$. As a result, we select the matrix W such that $GW = I_{k\times k}$, where I is the identity matrix. Then $h_i = u_i + (eW)_i$, in which $(.)_i$ denotes the i^{th} component. If noise is small enough, we will have $z_i = u_i$, which is exactly what we want.

On the other hand, if we pick $W' = G$, then if $\underline{z} = \underline{u}$, we will get $\underline{y} = \underline{x}$, which is the desirable association between $\underline{u}$ and $\underline{x}$. On the other hand, if there are some errors left in $\underline{z}$, i.e. $\underline{z} = \underline{u} + \underline{e'}$, then $\underline{y} = \underline{x} + \underline{e'}.G$ [1] Hence, this process of error correction continues until no errors are left or the network

[1]For the moment we forget about the upper bound on output of code nodes.

get stuck in a spurious state.

As a result, all we have to do is to carefully choose the generator matrix G such that code symbols generated by $\underline{x} = \underline{u}G$ are integers and bounded if message symbols u_i can take only two values *up* and *down* (both non-negative). Furthermore, G must be chosen such that its pseudo-inverse W has small norm so that $\underline{e}W$ is small enough not to cause any problem in $h_i = u_i + (eW)_i$.

In the next section, we show that often a random full rank G is sufficient for this purpose. However particular choices of G will improve the results significantly.

4 Simulation Results

In this section, we present the simulation results on using a lattice-like mechanism to perform association tasks. As mentioned before, we generate codewords $\underline{x}$ according to $\underline{x} = \underline{u}G$ where all operations are done in real field. Elements of the message vectors $\underline{u}$ can take two values: *up* and *down*. Our goal is develop an association between a message vector $\underline{u}$ and its corresponding codeword $\underline{x} = \underline{u}G$. We consider all possible 2^k message vectors and would like store all of them safely. We fix the weight matrices as explained in the previous section, i.e. the weight matrix from code nodes to message nodes (denoted by W) is chosen such that $GW = I_{k\times k}$. The reverse weight matrix W' from message nodes to the code nodes is equal to the generator matrix, i.e. $W' = G$. The neural threshold for message nodes is the middle of *up* and *down* values, i.e. $\theta_i = (up + down)/2$, for all z_i.

We initialize the code nodes with a noisy version of the codewords and let the network evolve until it reaches a stable point. If in the stable state $\underline{z} \neq \underline{u}$, we declare an error. For a given generator matrix, 100 noisy codewords are considered and the number of errors is counted. For error model, we consider a rounder Gaussian, i.e. we consider a Gaussian random variable r with mean zero and variance σ_n^2. Then $e_i = round(r)$. Furthermore, if e_i is too negative $y_i = x_i + e_i$ is truncated such that it is always greater than or equal to zero.

In the beginning, we consider a randomly generated G. We fix an average degree d_{ave} for G and generate G as follows: for each column of G, we pick d_{ave} positions and assign a random integer from 1 to g_{max} to these positions. Other positions would be set equal to zero. We generate 500 such random matrices and for each of them, generate 100 noisy codewords and investigate the performance of the method. The error probability would then be the average of the number of errors over these 50000 cases.

The value g_{max} controls the maximum value of codewords and the larger it is, the bigger codewords would be, which results in better error correction (since W would become smaller and $\underline{e}W$ will be small enough to let error correction happen). However, we can not increase g_{max} arbitrarily because it results in codewords becoming larger than their upper limits h_{max}.

Figure 2 illustrates the error rate as a function of SNR for $n = 15$, $k = 8$, $d_{ave} = 6$ and various values of g_{max}. Here, h_{max} is fixed to 120, $up = 2$, $down = 0$ and SNR is defines as the difference between up and $down$ powers divided by noise power, i.e. $SNR = (up - down)/\sigma^2$.[2] Note that in the figures error rate 0 is not illustrated because we are using loglog plots.

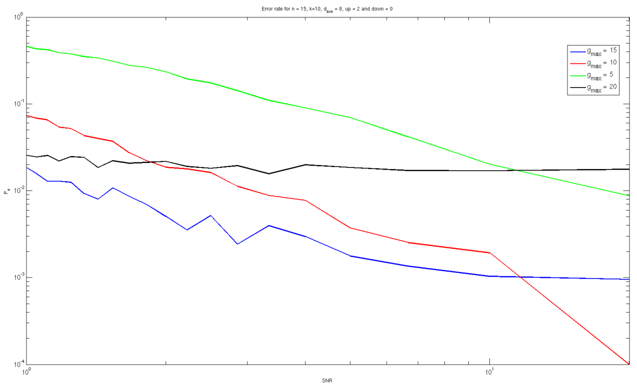

Figure 2: Error rate for $n = 15$, $k = 8$, $d_{ave} = 6$, $up = 2$ and $down = 0$

Figure 3 shows the error rate for the same setup as before but for $d_{ave} = 8$. Error rate for $n = 15$, $k = 8$ and $d_{ave} = 4$ is depicted in figure 4. Everything else is the same as before.

From the figures, it is obvious that there is an optimal choice for g_{max}. Lower values perform worse because of the smaller distances between code-

[2]Probably this definition of SNR is not the most accurate one as the signal power should be averaged over all symbols. Nevertheless, it gives a qualitative measure of performance against noise.

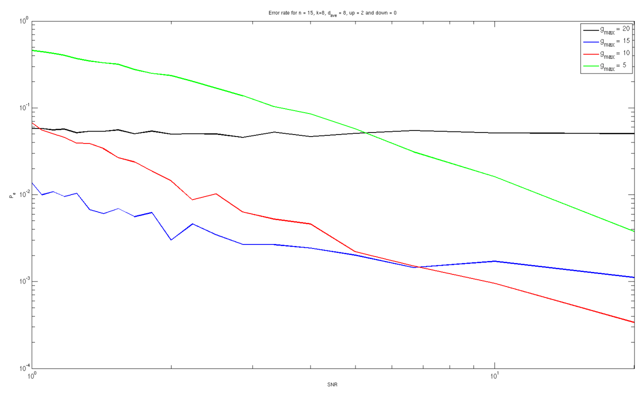

Figure 3: Error rate for $n = 15$, $k = 8$, $d_{ave} = 8$, $up = 2$ and $down = 0$

words and higher values would perform not well because of the upper bound h_{max} on the codewords values. The best value of g_{max} depends on d_{ave}. The sparser the matrix is the larger g_{max} can be. Furthermore, the optimal value of d_{ave} seems to be 6 for larger SNRs and 8 for lower SNRs.

So far, we have assumed that *up* and *down* values are both non-negative. However, if we allow negative values in our setup as well, the performance would improve dramatically. Figures 5 to 7 illustrates the error rate for $up = 2$, $donw = -2$ and the rest of parameters are the same as figures 2 to 4.

4.1 Choosing a Proper Generator Matrix

Now if we consider a particular choice of G instead of random constructions, we can get a better result. To do that, we generate G randomly for each g_{max} and pick the one with good error probabilities. We keep this G fixed for the rest of the simulations. Figures 8 and 9 illustrates the performance for $d_{ave} = 4$ and $d_{ave} = 8$, respectively. In all cases, $n = 15$, $k = 8$, $up = 2$ and $down = 0$. Note that for cases that error probability is zero over the simulations, the corresponding line is not depicted in the figures as loglog plots are used.

The same results when $up = 2$ and $donw = -2$ is illustrated in figures 10

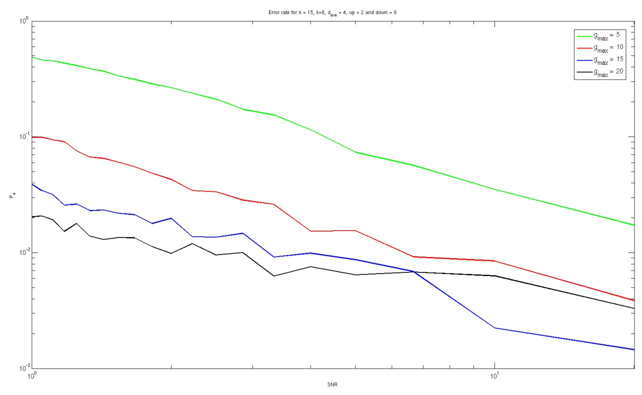

Figure 4: Error rate for $n = 15$, $k = 8$, $d_{ave} = 4$, $up = 2$ and $down = 0$

and 11.

One clearly sees the effect of g_{max}, as it is increased, performance gets better. Furthermore, it is obvious that for a proper choice of G, we get much better results. Hence, if we find a good G, we might get very nice results in term of error rate.

5 Conclusions and Future Works

In this report, we considered applications of methods similar to lattice codes in designing neural mechanisms of associative memory. We showed that for particular choices of code ensemble and neural thresholds, we can store all 2^k patterns with small error probability.

However, there are still some drawbacks in this approach: first of all, our choice of weight matrix involves pseudo-inverse calculations. Furthermore, we have not yet come up with a neural training rule which constructs the weight matrices as a function of patterns. One approach that could possibly resolve the first issue is to consider a parity check matrix (denoted by H) of the lattice code which has the property that if a correct codeword is given we get $H\underline{x}^T = 0$. Therefore, if there is no noise, the check nodes will not fire we stop. Otherwise, check nodes send feedback to the message nodes and

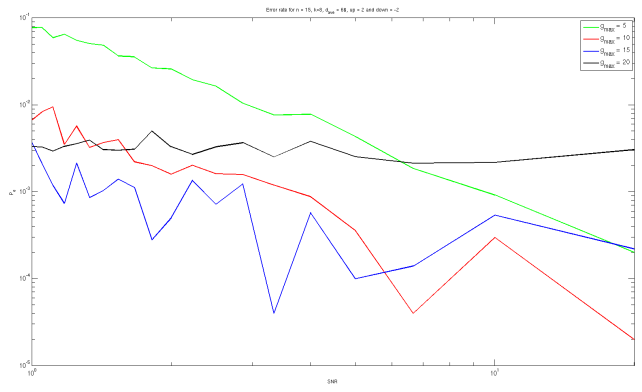

Figure 5: Error rate for $n = 15$, $k = 8$, $d_{ave} = 6$, $up = 2$ and $down = -2$

they act accordingly to correct the errors.

There are other ideas as well for future. The most important one is to include the inter-spike delay into the model as well. More specifically, we can discriminate between the two cases that two spikes were fired consecutively and the case that one of them was fired in the beginning of the time window and the other one in the end of the window. In the current model, both of these case is considered as the output level 2. However, if we consider the delay we can consider the output value as a fractional number whose integer part represents the firing rate and the fractional part represents the inter-spike delay between two consecutive spikes.

even better, we can use the inter-spike delay to represent signs. For instance, we can multiply the firing rate by a phase which is a function of the inter-spike delay and for certain values of the delay, we will get negative values. These negative values are important since as we saw in the previous section, allowing negative values would improve the performance significantly.

References

[1] J. J. Hopfield, "Neurons with graded response have collective computational properties like those of two-state neurons", Proc. National Acad.

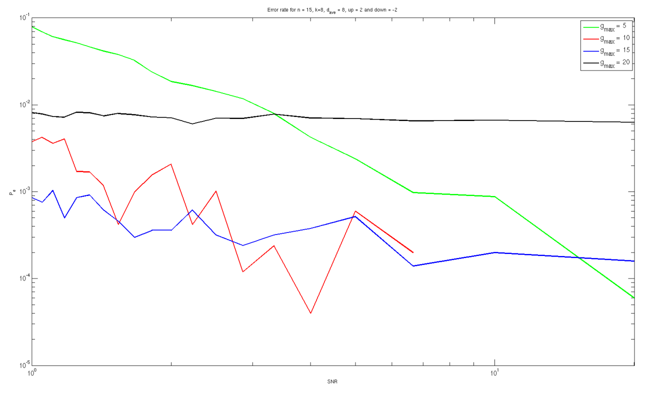

Figure 6: Error rate for $n = 15$, $k = 8$, $d_{ave} = 8$, $up = 2$ and $down = -2$

Sci., Vol. 81, No. 10, 1984, pp. 3088-3092.

[2] J. Hertz, A. Krogh, R. G. Palmer, "Introduction to the theory of neural computation", Addison-Wesley, 1991.

[3] R. McEliece, E. Posner, E. Rodemich, S. Venkatesh,"The capacity of the Hopfield associative memory", IEEE Trans. Inf. Theory, Vol. 33, No. 4, 1987, pp. 461-482.

[4] S. S. Venkatesh, D. Psaltis, "Linear and logarithmic capacities in associative neural networks", IEEE Trans. Inf. Theory, Vol. 35, No. 3, 1989, pp. 558-568.

[5] M. K. Muezzinoglu, C. Guzelis; J. M. Zurada, "A new design method for the complex-valued multistate Hopfield associative memory", IEEE Trans. Neur. Net. Vol. 14, No. 4, 2003, pp. 891-899.

[6] N. Sommer, M. Feder, O. Shalvi, "Low Density Lattice Codes", IEEE Trans. Inf. Theory, Vol. 54, No. 4, 2008, pp. 1561-1585.

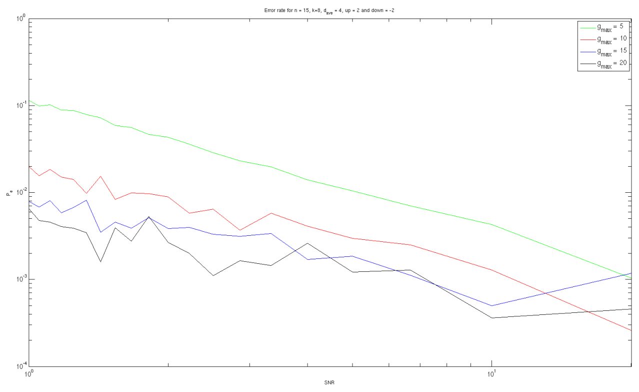

Figure 7: Error rate for $n = 15$, $k = 8$, $d_{ave} = 4$, $up = 2$ and $down = -2$

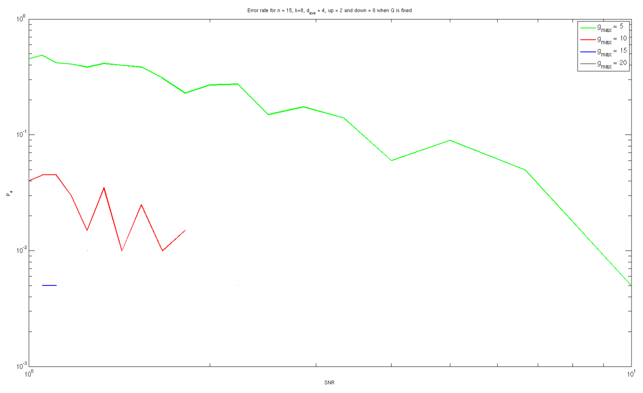

Figure 8: Error rate for $n = 15$, $k = 8$, $d_{ave} = 4$, $up = 2$ and $down = 0$ when G is fixed

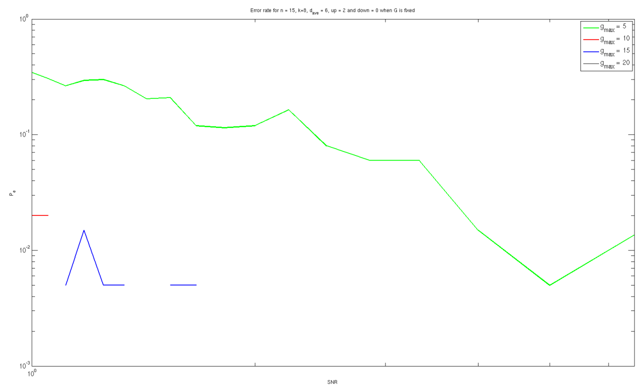

Figure 9: Error rate for $n = 15$, $k = 8$, $d_{ave} = 6$, $up = 2$ and $down = 0$ when G is fixed

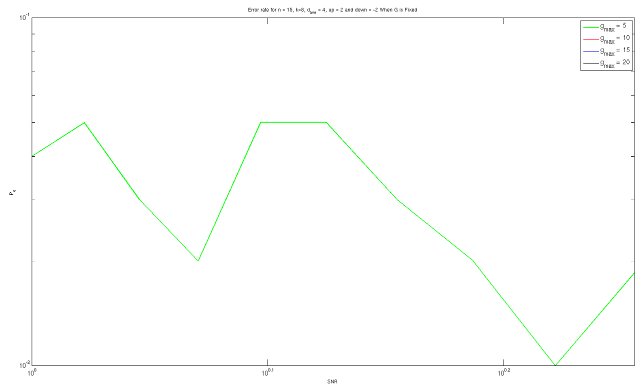

Figure 10: Error rate for $n = 15$, $k = 8$, $d_{ave} = 4$, $up = 2$ and $down = -2$ when G is fixed

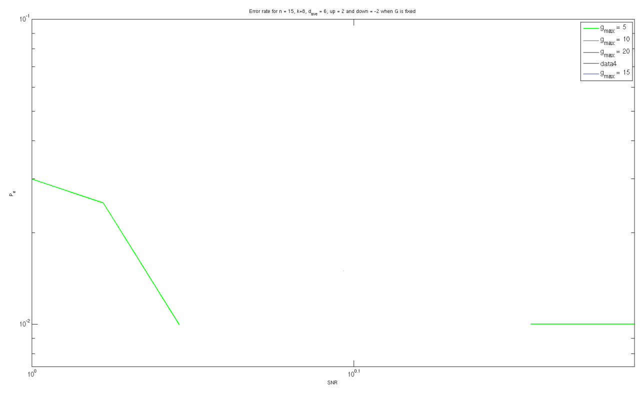

Figure 11: Error rate for $n = 15$, $k = 8$, $d_{ave} = 6$, $up = 2$ and $down = -2$ when G is fixed