Attractor Neural Networks for Modelling Associative Memory

Wael Al Jishi
wa910@imparial.ac.uk

Niklas Hambuechen
nh910@imparial.ac.uk

Razvan Marinescu
rvm10@imperial.ac.uk

Mihaela Rosca
mcr10@imparial.ac.uk

Lukasz Severyn
lk1110@imparial.ac.uk

January 2013

Executive Summary

Attractor Neural networks have been a subject of intensive research in the past 30 years. They have been proposed as a model of associative memory and have been used in ranging from facial and speech recognition to modelling biological activities of the human brain.

The aim of our project was to attain a deeper understanding of these networks by running various simulations and investigating their properties. For this purpose, we have demonstrated their importance by implementing an image recognition software. Furthermore, we have also used them as an aid for modelling a psychological concept known as Attachment Theory, in order to better understand the human mind and how to cure various mental disorders.

Our project attempts to model, at a metaphorical level, an individual's emotional mindset, and how it is formed due to the influences of experiences, which manifest as memories, encountered in early life. We have implemented several mathematical models of neural networks, that can simulate various functions of the brain. Amongst these, two of them are of uttermost importance: learning and recalling information.

We have used our networks to analyse and quantify the influence of memories, analysing factors affecting their probability of being recalled, or how they can be either forgotten or reinforced. The latter is extremely important for the psychological studies, since it paves the way for curing some mental disorders whose roots lie in negative experiences encountered early in life.

Contents

Chapter 1

Introduction

1.1 Motivation

The human brain is considered to be one of the most complex objects which have puzzled scientists and philosophers alike throughout the centuries. To gain an understanding of how this mysterious black box works, through experimentation and modelling, would be of great aid to the furthering of science and humanity alike. A part of this includes developing cures for a multitude of mental and brain-related disorders.

Several medical discoveries from the 20th century have enlightened our knowledge of the human brain. One consequence of this was the development of different mathematical models of artificial neural networks, inspired by the fields of biology and medicine.

Our motivation is to use some of these neural networks to explore the mathematics of a developing theory called Self-Attachment. Part of Attachment theory, this strategy aims to help cure various mental problems that people currently facing. Recent research has shown that a mathematical methodology of analysing these disorders is starting to become feasible. [13]

The subsequent chapters will introduce the Attachment Theory and the mathematical model. We have mainly focused on the technical side, and have described psychological analogies that place our technical results in context of the theory.

1.2 Objectives

Our main objective is to confirm the results of previous work done by Federico Macinelli, who has been analysing the attachment theory using neural networks. He has performed several experiments regarding clusters of attractors, basin sizes, and Gaussian-distributed patterns. Furthermore, we have been aiming at extending his results by exploring the following concepts:

- Using the network for performing image recognition
- Restricted Boltzmann machines
- Super-attractors

In addition to that, we were aiming to improve some of the methods that were used in his experiments. This includes the technique for sampling Gaussian-distributed patterns or for calculating basin sizes. Our final aims consisted of explaining some of the subsequent inconsistencies that have been found in his results.

1.3 Our achievements

Since our project was related to exploring attractor-based neural networks, we have obtained interesting results about their attractors. These attractors are analogous to learned memories or experiences that an individual has learned.

Our main contributions are outlined below:

- Confirming Federico's results by reimplementing them in a completely different environment (Haskell)
- Proving that the Hopfield network is capable of performing image recognition, by learning image patterns and then recalling the closest image, when queried for an input. Furthermore, we have proven it's associative memory properties, by successfully recalling some of the learned images.
- We found out that training patterns are not guaranteed to become fixed points in the Hopfield network. This is an aspect that was not mentioned in similar research papers, and we first thought the the patterns are always fixed points.
- Extending the research to encompass the Boltzmann Machine, which provides a nice way of overcoming some limitations of the Hopfield Network. Amongst other features, it can prevent convergence to spurious patterns by using a stochastic update rule.
- A thorough analysis of Super-Attractors, which represent patterns that have been used multiple times in the training process. They generally have greater basin sizes, and therefore patterns will have a greater chance of converging to them compared to normal attractors.

Chapter 2

Background Research

This chapter details the background research we have done in order to understand the Hopfield Neural Networks, realise their importance in various applications and to investigate the feasibility of our project.

2.1 Attachment Theory

One of the most interesting applications of the Hopfield Networks and Boltzmann Machines is related to modelling the Attachment Theory. This is a psychological study that describes the dynamics of long-term relationships between humans[15]. It focuses primarily on the type of attachment that an infant develops with the primary caregiver. There exist 2 main types of attachment: secure and insecure. Insecure patterns are further divided into insecure-avoidant, insecure-disorganised and insecure-resistant. The type of attachment developed by infants depends on the quality of care they have received[15].

2.1.1 Strange Situation Procedure

The classification of infants into the corresponding attachment type is formulated in the Strange Situation Procedure. The child is observed playing for 20 minutes while caregivers and strangers enter and leave the room, recreating the environment of the familiar and unfamiliar presence in most of the children's lives.[14]

2.2 Practical applications of Attractor Neural Networks

Neural Networks have been successfully used in medical computing, in order to diagnose Parkinson's disease[3]. Other applications include facial recognition, which we have implemented for the purpose of this project, combinatorial problems such as the Travelling Salesman[5]. One practical application of the attractor neural networks, such as the Boltzmann machine, is the performance improvement of speech recognition software[1].

2.3 Hopfield Networks

The Hopfield Networks are a form of recurrent artificial neural networks invented by John Hopfield in 1982 [17]. It aims to store and retrieve information like the human brain. It basically consists of a complete graph of N neurons, each having a value of +1 or -1 associated to it. Since the graph is complete, each pair of neurons (i,j) is connected by an edge, having an associated weight w_{ij}.

2.3.1 Updating the Hopfield Network

The network is able to update the values associated to neurons by adhering to certain simple rules. Updating can be performed in two different manners:

- Synchronous: All nodes are updated at a time. This requires a central clock to the system in order to maintain synchronisation. This method is less realistic, since biological or physical systems lack a global clock that keeps track of time.
- Asynchronous: Only one node is updated at a time. A random node can be chosen as the next one to get updated, or otherwise a sequence of nodes can be imposed a-priori.

Assume N neurons = 1.. N, with values $x_i = \pm 1$. The updating of an individual node i is performed by first calculating a weighted sum of the neighbouring nodes, and then applying the sign function:

$$x_i = sgn(\sum_{j=1}^{N} w_{ij}x_j + b_i)$$

where b_i is a bias that we will consider to be equal to 0 for the purpose of this project.

Suppose the weight w_{ij} between neurons i and j is positive. Then, if the value of x_j is positive, then the term $w_{ij}x_j$ will also be positive, and will drag the linear sum value to a positive value. This would mean that the neuron x_i would also be dragged towards a positive value.

If updating is repeatedly performed, the network would eventually may converge to an attractor pattern under asynchronous updating, or it might sometimes cycle under synchronous updating.

2.3.2 Training using the Hebbian Rule

The Hebbian theory has been introduced by Donald Hebb in 1949, in order to explain "associative learning", in which simultaneous activation of neuron cells leads to pronounced increases in synaptic strength between those cells [16]. It is often summarised as "Neurons that fire together, wire together".

For the Hopfield Networks, this is implemented in the following manner, when learning p patterns:

$$w_{ij} = \frac{1}{N}\sum_{\mu=1}^{p} \epsilon_i^\mu \epsilon_j^\mu$$

For pattern μ, if the bits corresponding to neurons i and j are equal, then the product $\epsilon_i^\mu \epsilon_j^\mu$ will be positive. This would, in turn, have a positive effect on the weight w_{ij} and the values of i and j will tend to become equal. The opposite happens if the bits corresponding to neurons i and j are different.

2.3.3 Spurious Patterns

Patterns that the network uses for training, called retrieval states, become attractors of the system. Repeated updates would eventually lead to convergence to one of the retrieval states. However, sometimes the network will converge to spurious patterns, that are different from the training patterns.

A a linear combination of an odd number of stored patterns, in this case 3 patterns, would give us a spurious state:

$$\epsilon_i^{mix} = \pm sgn(\pm\epsilon_i^{\mu_1} \pm \epsilon_i^{\mu_2} \pm \epsilon_i^{\mu_3})$$

However, we shall later on see that the probability of the network converging to these states is small, because of the tiny basin size. Furthermore, the Boltzmann Machine has the capacity of getting out of these spurious states, because of the probabilistic updating of the nodes.

2.3.4 Energy Landscape

One of Hopfield's most important contributions was to associate a Lyapunov, or energy function to the neural network. This is calculated as:

$$E = -\frac{1}{2}\sum_{i,j=1}^{N} w_{ij}x_i x_j$$

The function decreases as the system gets updated until it reaches a local minimum, corresponding to an attractor (see fig. 2.1).

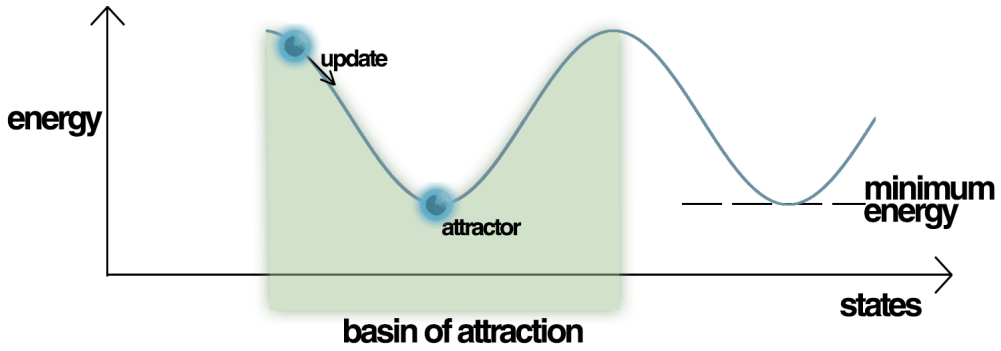


Figure 2.1: Energy Landscape highlighting a current state of the network (up the hill), an attractor to which it will eventually converge, a minimum energy level and a basin of attraction filled in green. Note how the update of the Hopfield Network is always going down in Energy. The basin of attraction concept will be introduced later on.

2.3.5 Stability and Capacity

Concepts of stability and capacity are important in the analysis of Hopfield Networks. A neuron unit ϵ_i^{μ} is said to be stable if the update rule will not change its state. The mathematical condition for this is:

$$x_i = sgn(\sum_{j=1}^{N} w_{ij}x_j)$$

Since we are using the Hebbian Rule, we can replace w_{ij} with $\sum_{\nu} \epsilon_i^{\nu}\epsilon_j^{\nu}$, resulting in:

$$h_i^{\mu} = \frac{1}{N}\sum_{j}\sum_{\nu} \epsilon_i^{\nu}\epsilon_j^{\nu}\epsilon_j^{\mu}$$

We will now isolate the contribution of neuron i, in order to get:

$$h_i^{\mu} = \epsilon_i^{\nu} + \frac{1}{N}\sum_{j}\sum_{\nu\neq\mu} \epsilon_i^{\nu}\epsilon_j^{\nu}\epsilon_j^{\mu}$$

Now, the input for node i is made of two components: the first component depends on the node i itself, while the second component, called the **crosstalk term**, is directly dependent upon the neighbours of the node.

If we multiply the crosstalk term by $-\epsilon_i^\nu$, we get a quantity that will help us study the capacity:

$$C_i^\nu = -\epsilon_i^\nu \frac{1}{N} \sum_j \sum_{\nu \neq \mu} \epsilon_i^\nu \epsilon_j^\nu \epsilon_j^\mu$$

If C_i^ν is negative, then the cross-talk term has the same sign as unit i, and thus this value will not change [2]. However, if C_i^ν is positive and greater than 1, then ϵ_i^ν will change, so node i will become unstable. This quantity can be used to show, with further analysis, that the capacity of the Hopfield Network is revolving around the value 0.138 (approximately 138 **random** patterns can be recalled for every 1000 neurons) (Hertz et al., 1991).

2.3.6 Training using the Pseudo-Inverse Rule

The pseudo-inverse rule makes use of the inverse of a matrix in order to perform the learning process. In contrast to the Hebbian Learning, it offers a higher capacity and performs better with correlated, linearly independent patterns. The weight matrix is computed as:

$$w_{ij} = \frac{1}{N} \sum_{\mu\nu} \epsilon_i^\mu Q^{-1} \epsilon_j^\mu$$

In this case, Q is the overlap matrix: $Q_{\mu\nu} = \frac{1}{N} \sum_i \epsilon_i^\mu \epsilon_i^\nu$

Now that we have introduced a second learning rule, it is worth mentioning a few desirable properties that neural network learning rules should aim to have. In Artificial Neural Networks, learning rules can be:

- Local: each weight is updated using information available to neurons on either side of the connection.
- Incremental: new patterns can be learned without using information from the older patterns. When a new pattern is used for training, the new values for the weights only depend on the old values and on the bits of the pattern.

These properties are desirable, since learning becomes more biologically plausible. For example, since our brain is always learning new concepts, we can reason that learning is incremental. A learning system that would not be incremental would generally be trained only once, with a huge batch of training data.

It is important to mention that Hebbian Learning is both local and incremental, whereas the Pseudo-inverse learning rule is neither local nor incremental, since it depends upon the computation of an inverse matrix that contains information about all the patterns together.

2.3.7 Training using the Storkey Rule

The weight matrix of an attractor neural network is said to follow the Storkey learning rule if it obeys:

$$w_{ij}^{\nu-1} = w_{ij}^{\nu-1} + + \frac{1}{n} \epsilon_i^\nu \epsilon_j^\nu - \frac{1}{n} \epsilon_i^\nu h_{ji}^\nu - \frac{1}{n} h_{ij}^\nu \epsilon_i^\nu$$

where $h_{ji}^\nu = \sum_{k=1, k \neq, j}^n w_{ik}^{\mu-1} \epsilon_k^\mu$ is a form of *local field* [10] at neuron i.

This rule is local, since the synapses take into account only neurons at their sides. This rule is slightly more powerful than the generalised Hebb rule, since it makes use of more information. Because of the action field, each neuron uses information from all the neighbours. As Storkey has showed in 1997, the capacity of the network trained with this rule is greater than compared to Hebbian rule [10].

2.4 Restricted Boltzmann Machines

Hopfield networks are deterministic: training a network with the same patterns will yield the same weights, and matching a pattern against the network will always give the same result. This has the advantage of simplicity and ease of testing. However, as mentioned before, there are spurious attractors in the Hopfields network (linear combination of an odd number of training patterns). Using stochastic updating rules will decrease the probability of being 'stuck' in such a state. This resembles simulated annealing, but ensuring we are not caught in a local minima (in the energy landscape).

Restricted Boltzmann Machines have been used for various purposes in recent years [7] [12] [9] , most of it conducted by Geoffrey E. Hinton at university of Toronto, Canada. They have a simple structure: one layer of visible units and one layer of hidden units, which form a bipartite graph, as in Figure. The patterns used to train the network correspond to the visible (exposed units), while the hidden units correspond to the attributes of the features we would like to learn. It is worth mentioning that in the most common and simple form, both the visible and hidden units take binary values. This is the approach we also adopted for our implementation.

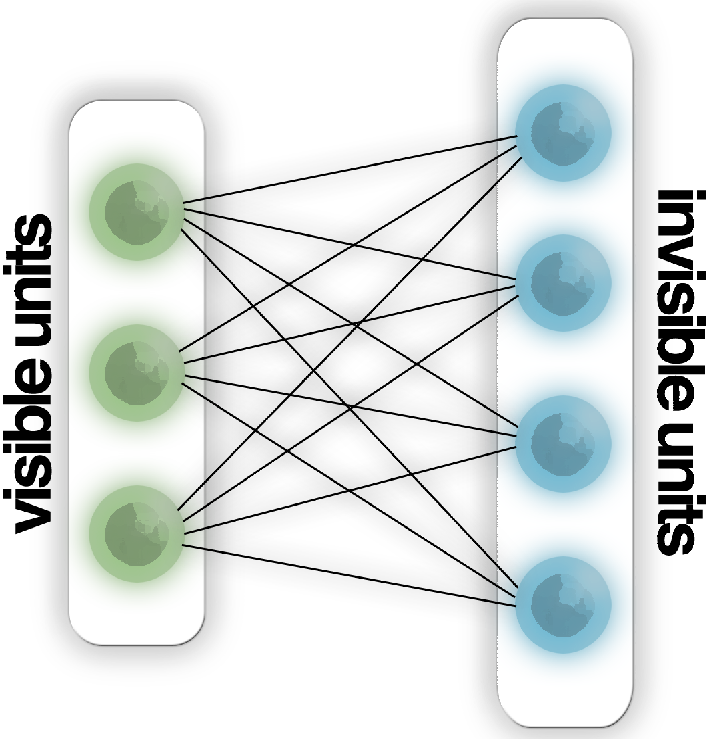


Figure 2.2: Restricted Boltzmann Machine

The connections between the neurons in the two layers is symmetrical, so it can be represented as a weight matrix which keeps the connection between hidden and visible layers. A state of a Restricted Boltzmann machine is given by the values of both the visible and given state. The corresponding Hopfield energy formula for a state is given by :

$$E(v,h) = -\sum_{i \in V} a_i v_i - \sum_{j \in H} b_j h_i - \sum_{i \in V, j \in H} v_i h_j w_{\text{ij}}$$

where, $w$is the weight matrix, a and $b$are the vectors of biases corresponding to the visible, respectively hidden layers. As expected, $v$is vector of visible units and h is the vector of hidden

units.

We denote by V = {1, .. length v} , the indices of a visible vector and by H = {1, .. length h} the indices of a hidden vector.

2.4.1 Probability of a state

After traning, the network assigns a probability to each possible state, as follows:

$$p(v,h) = \frac{1}{Z} e^{-E(v,h)}$$

where Z is used to normalize the probabilities

$$Z = \sum_{v,h} e^{-E(v,h)}$$

Thus, the probability the network asessings for a visibile vector v:

$$p(v) = \sum_{h} \frac{1}{Z} e^{-E(v,h)} \tag{2.1}$$

2.4.2 Training the network

The network can be trained such that one maximizes the probability of a training pattern. The derivative of the log probability of a training vector (given by (1)), is simple, as follows:

$$\frac{\delta \log p(v)}{\delta w_{\text{ij}}} = < v_i h_i >_{\text{data}} - < v_i h_i >_{\text{model}}$$

Due to the fact that the Restricted Boltzmann Machine can be represented as a bipartite graph, it is easy to attain an unabiased sample of the hidden units, given the visible units.

$$p(h_j = 1 \mid v) = \sigma \left(b_j + \sum_{i \in V} v_i w_{\text{ij}} \right) \tag{2.2}$$

For the visible units, the same formula gives a biased sample of the visible units.

$$p(v_i = 1 \mid h) = \sigma \left(a_i + \sum_{j \in H} h_j w_{\text{ij}} \right) \tag{2.3}$$

where $\sigma = \frac{1}{1+e^{-x}}$, is the logistic sigmoid function.

There are ways of getting an unbiased sample of the visible units, but they are very time consuming. In practice, the contrastive divergence algorithm is used as a faster substitute. The visible units are set to a training vector and the binary states of the hidden vector are computed using (2). These binary units are used to reconstruct the states of the visible vector using (3).

The training rule then becomes:

$$\Delta w_{\text{ij}} = \lambda \left(< v_i h_i >_{\text{data}} - < v_i h_i >_{\text{reconstruction}} \right)$$

We note that the above training rule has the desired properties for modeling the biological brain: it is both local and incremental. Angle brackets denote the expected value under the given distribution by the following subscript.

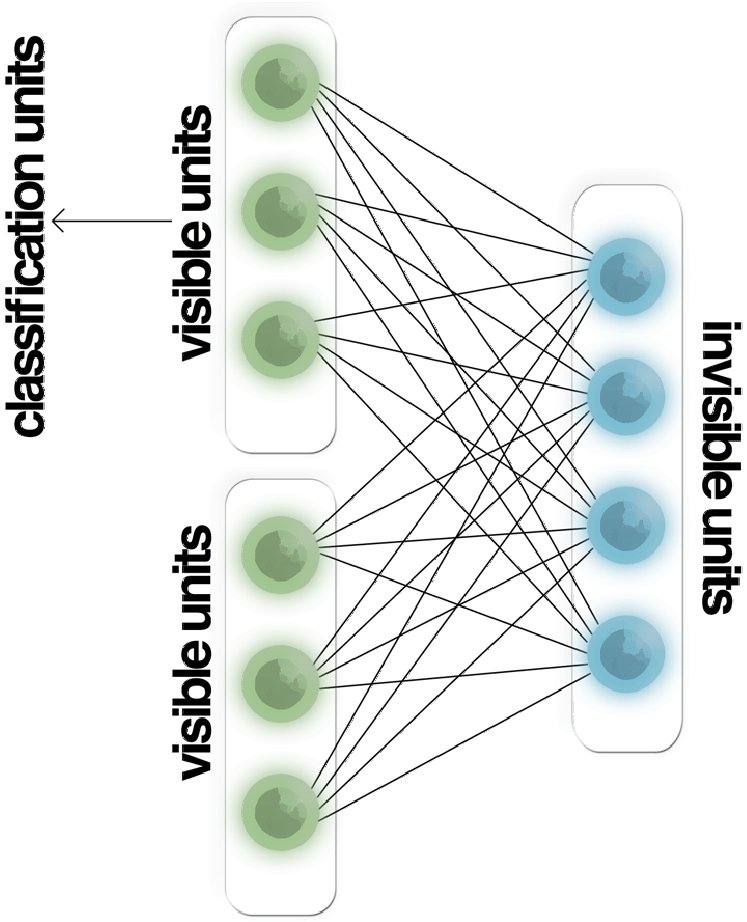


Figure 2.3: Classification Boltzmann machine

2.4.3 Using Restricted Boltzmann Machines for classification

As suggested by Hinton in section 16 [4], there are various ways of using Restricted Boltzmann Machines for classification. We have employed two methods, one which we developed ourselves and another one described in the paper. The latter is adding another set of visible units, called the *classification units*.

The vector of hidden units is used to model the joint distribution between a vector of inputs v and a target (classification) vector y. The classification vector y corresponds to a class. It's length is given by the number of classes. $y = e_c$, where c is the class the vector is modeling (e_c is a vector with all zeros, expect from the position c, where it is 1).

As seen from Figure 2, one now needs to weight matrices, which we will denote by W(the weight matrix between the visible units and hidden ones) and U (the weight matrix and between the classification units and hidden ones). Vectors b, c, d correspond to the vector of biases for the visible units (training patterns), hidden units and classification units, respectively.

The energy function for this model is:

$$E\left(v, y, h\right) = -d^T y - c^T h - b^T v - h^T W v - h^T U y$$

Which gives rise to the following formula for the probability of a configuration v, y.

$$p\left(v, y, h\right) = \frac{e^{-E(v,y,h)}}{Z}$$

where Z is the normalizing constant. Thus,

$$p(v,y) = \sum_h p(v,y,h) = \sum_h \frac{e^{-E(v,y,h)}}{Z}$$

The distribution used for the Classification Boltzmann Machine are the following:

$$p(h_i = 1 \,|v,y\,) = \sigma\left(c_j + W_j v + U_j y\right) \tag{2.4}$$

$$p(v_i = 1 \,|h\,) = \sigma\left(b_j + h^T W^i\right) \tag{2.5}$$

$$p(y = e_c \,|h\,) = \frac{e^{d^T e_c + h^T U e_c}}{N} \tag{2.6}$$

where N is the normalizing constant $\sum_c e^{d^T e_c + h^T U e_c}$.

Note that we denote by W_i the row i of matrix W, and by W^i the column i of matrix W.

Valuable reference for this was given to us from [7] and [6].

2.4.4 Learning

Contrastive divergence can be used to train the network, giving rise to the following update rules:

$$\begin{aligned}
b' &= b + \lambda\left(v - v_{\text{sampled}}\right) \\
c' &= c + \lambda\left(h_\sigma - h_{\sigma\,\text{sampled}}\right) \\
d' &= d + \lambda\left(y - y_{\text{sampled}}\right) \\
W' &= W + \lambda\left(h_\sigma v^T - h_{\sigma\,\text{sampled}} v_{\text{sampled}}^T\right) \\
U' &= U + \lambda\left(h_\sigma y^T - h_{\sigma\,\text{sampled}} y_{\text{sampled}}^T\right)
\end{aligned}$$

where we denote by x' the new value of parameter x. $v_{\text{sampled}}$and $y_{\text{sampled}}$are obtained by sampling the distributions in (5) and (6).

$$\begin{aligned}
h_\sigma &= \sigma\left(c_j + W_j v + U_j y\right) \\
h_{\sigma\,\text{sampled}} &= \sigma\left(c_j + W_j v_{\text{sampled}} + U_j y_{\text{sampled}}\right)
\end{aligned}$$

2.4.5 Classification

$$p(y = e_c \,|v\,) = \frac{e^{-F(v,e_c)}}{\sum_d e^{-F(v,e_d)}}$$

where $F(v,e_c)$is the free energy function

$$F(v,e_c) = -d^T y - \sum_{j=1}^{H} s\left(c_j + W_j v + U_j y\right)$$

where $s(x) = \log\left(1 + e^x\right)$

The implementation of the Classification Boltzmann Machine can be found in `ClassficationBoltzmannMachine.hs`.

2.4.6 New method

Another method we have employed using Boltzmann Machines was created by us. We have never seen this approach used somewhere else. Instead of creating 2 types of visible units, we use the simple Restricted Boltzmann Machine, with one type of visible units (and hence a single matrix). For each training vector we append the classification at the end. The classification is represented as a binary vector, either as above, by using e_c, where c is the classification of the current pattern, or even in a more compressed manner, by creating the binary vector using the representation in base 2 of class c.

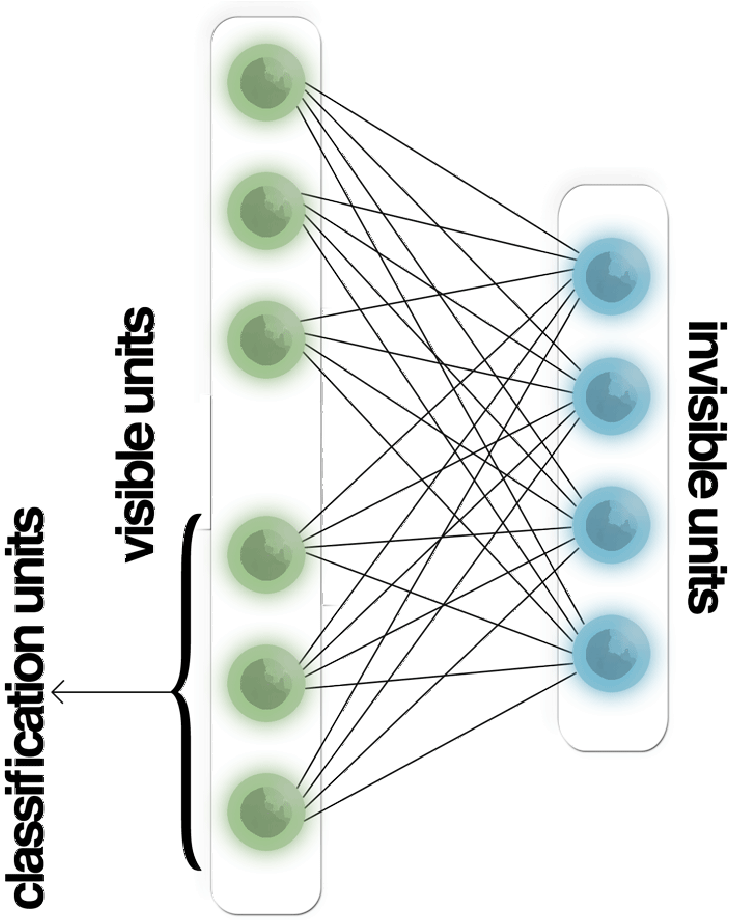


Figure 2.4: Boltzmann machines according to our new method

The training is done in the usual way, but with the complete vectors (actual pattern and classification). In our case, as different patterns ought to have different classifications, we use as class the index of the pattern in the list (with removed duplicates) of training patterns.

When a pattern needs to be matched to one of the training patterns for recognition, one uses the log probability to compute the probability of each of the classifications, and choses the one with maximum probability.

As given in [4], the log probability is given by:

$$\log\left(\text{class} = c \,|v\,\right) = \frac{e^{-F_c(v)}}{\sum_{c'} e^{-F_{c'}(v)}}$$

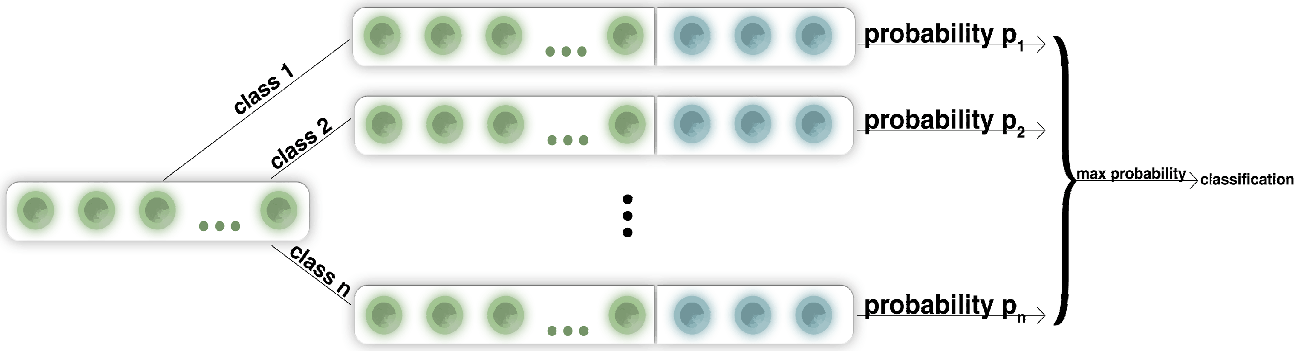


Figure 2.5: The recognition process in the new Boltzmann machine.

$$F(v) = -\sum_{i \in V} v_i a_i - \sum_{j \in H} \log\left(1 + e^{x_j}\right)$$

where $x_j = b_j + \sum_i w_{ij} v_j$
The implementation of this procedure can be found in `RestrictedBoltzmannMachine.hs`.

Chapter 3

Exploring The Hopfield Model

We started out the analysis by experimenting with converge of patterns. We shall present two experiments, that provide some interesting insight for the reader into the Hopfield Model.

3.1 Facial Recognition for Police criminal records

This experiment describes how the Hopfield network might be used by a Police Office, in order to identify criminals. Since the police has a big database that contains facial images of criminals, the network can perform identify, given an image, which criminal is depicted.

Before being sent to the Hopfield model, the image suffers a few normalizing transformations:

1. the background is completely removed
2. the image is scaled to 25x25 pixels.
3. conversion first to grayscale, and then to only black or white pixels takes place

In figure 3.1, we present the convergence series for the input image of a criminal. The network used for this experiment had 400 neurons, running on images of 20x20 pixels. It clearly illustrates the how efficient the networks can be in these scenarios. However, the network might also convert to to spurious patterns, which might create problems in certain situations. See section 2.3.3 for a more detailed description.

Figure 3.1: Convergence sequence for a criminal in the police database

Furthermore, we have developed a GUI, powered by the underlying recognition core, to provide a friendly user experience, in addition to providing interactive facilities (figure 3.2). An overview of our the recognition system can be found in figure 3.3.

Figure 3.2: GUI for a criminal record recognition system.

3.2 Further analysis

Our initial aim was to explore various properties of the attractors in a Hopfield Network. We are interested in studying, testing or challenging several ideas about the neural network:

- Clusters of attractors: We are interested to see how the Hopfield network behaves when learning similarly correlated patterns. This implies training the network with a cluster of attractors, which will lower the capacity of the network. We believe that basin sizes will get smaller as the patterns get closer to each other.
- Super-Attractors: Find out what happens if the network is trained several times with the same pattern. We believe this super-attractor will have a larger basin size compared to the other attractors.
- Convergence: We plan to train the network with different types of attractors and find out to which ones patterns tend to converge.

3.3 Basins of Attraction

A basin of attraction for a particular attractor α is defined as the set of all states that will eventually converge to α, under repeated update. Here, we are particularly interested in finding a way to measure the size of such a basin of attraction. A large basin size would provide stability for the attractor, since the states in the neighbourhood would converge to it. A small basin size for an attractor would mean that the network might never recall the pattern corresponding to that attractor.

3.3.1 Measuring the Basin Size using the Storkey-Valabregue technique

A large part of our experiments were thus dedicated to measuring basin sizes of different attractors, clusters of attractors and a newly introduced concept of *Super-Attractors*, in section 3.7.

A recognised scientific method for calculating the basin size is the Storkey-Valabregue measurement. This can be computed as follows:

1. Initially n = 1

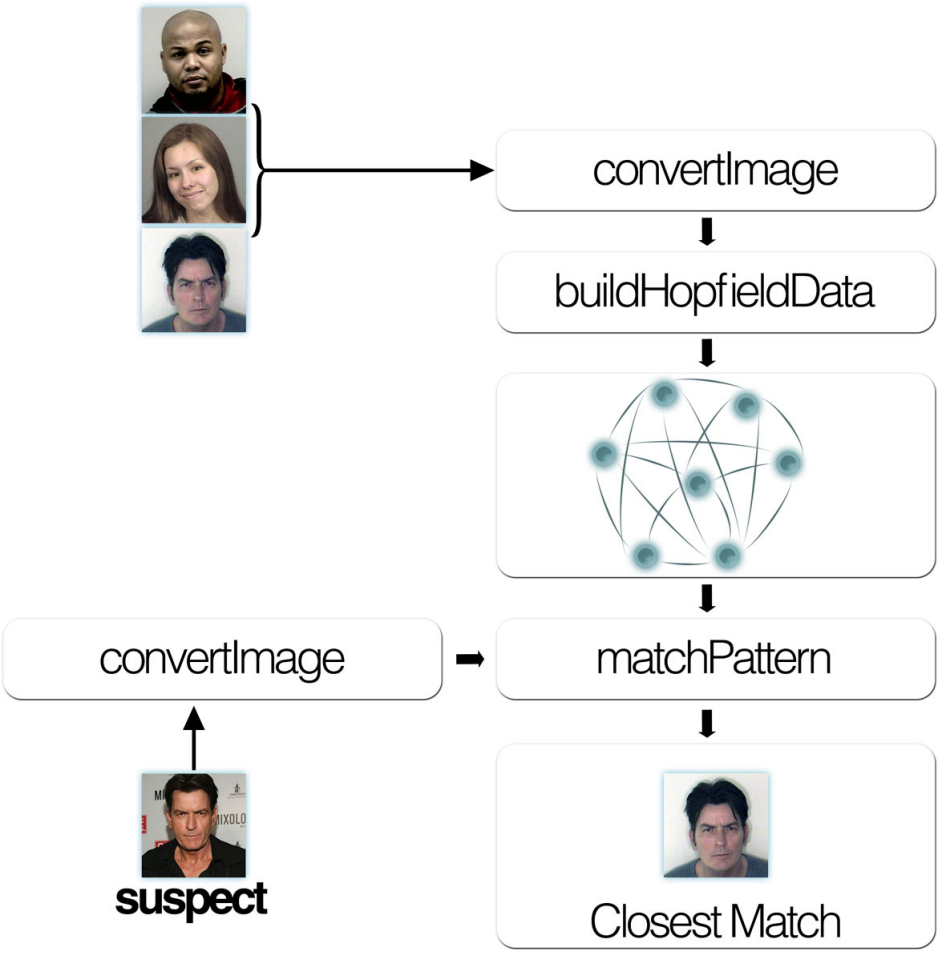


Figure 3.3: Outline of the Recognition System demonstrating how the various system components interact

2. Choose an initial fixed point corresponding to a stored pattern μ
3. Choose some initial normalised Hamming radius $r = r_0$
4. Let the set A be all the states Hamming-distant nr from the fixed point
5. Sample 100 states from A
6. Calculate how many of these states are attracted to the fixed point. Denote this number $t_\mu(r)$
7. If $t_\mu(r)$ is smaller than 90, stop and return nr
8. Increment n by a suitable amount and repeat from (3)
9. Repeat for each attractor.

3.4 Generating Clusters of Attractors

There are various ways of generating clusters of attractors (i.e. attractors that have a low Hamming distance between each other). We shall present two different methodologies that can be used to generate them.

One way is to start with a root pattern and then reverse each bit with a probability p. The new patterns can be interpreted as noisy versions of the root pattern. We shall denote this procedure T1.

3.4.1 Generating Gaussian-Distributed Clusters

Another objective of this work was to analyse patterns that are extracted according to a Gaussian distribution with a specific mean and variance.

Since the state space is 2^N for a network of N neurons, the Gaussian Distribution will have to be defined over a huge set of states. This is highly non-trivial, since we are dealing with binary patterns, that cannot be ordered in an easy way.

In this case, we will use Federico's method for sampling Gaussian distributed patterns [8, p. 33]. We tackle this issue using the simplest possible approach: to draw numbers from a Normal Distribution and then translate them into patterns that will preserve the distance between them. Formally, if x and y were drawn and $|x - y| = \delta$, then the Hamming distance between the encoded patterns x' and y' would also be $d(x', y') = \delta$.

Encoding of the patterns

In order to encode a number into a binary pattern, we first round the number to the nearest integer. Let k be the integer obtained as such. Now, from left to right, we set to 1 all the bits in locations 1..k of our pattern. The remaining bits are set to -1.

For example, if the pattern size is 7 and the integer obtained is 4, we get the following pattern: [1,1,1,1,-1,-1,-1].

Limitations of the encoding

Although the method has a big advantage for simplicity, we can only obtain N different patterns out of all 2^N patterns available in the state space. However, we can regain capacity if we start flipping bits, while at the same time keeping constant the hamming distance to some certain mean pattern μ.

Another idea would be to extend our distribution to a multi-variate distribution. In this case, the capacity of the network would increase from N to $\frac{n}{k}^k$, where k is the number of dimensions.

Unfortunately, we did not have enough time to implement these ideas, so we only experimented with the naive encoding of patterns described above.

3.5 First experiments with basin sizes

In this section we shall introduce the experiments that we used to analyse various properties of the Hopfield Network. We remind the reader about the two methods that we are going to use for generating clusters of attractors:

- T1: We start with a root pattern μ, and generate patterns by flipping each bit from μ with probability p.
- T2: This method is generating Gaussian-distributed patterns. We sample several numbers from a normal distribution with a certain mean and variance, and then encode each number into patterns of the form [1,1,1,-1,-1].

3.5.1 Basin Size for One Cluster using T1

Our first experiment is showing us the basin sizes for increasing values of p, the probability of flipping a bit. Method T1 is used, for a Hopfield Network of N neurons.

I. We generate a random pattern μ

II. For all values of probability p from 0 to 0.5

1. Starting from μ we generate P patterns using T1, and give them to the Hopfield network in order to be learned according to Hebb or Storkey rule.
2. We measure the basin size for all the patterns learned using the Storkey-Valabregue measurement and take their mean.

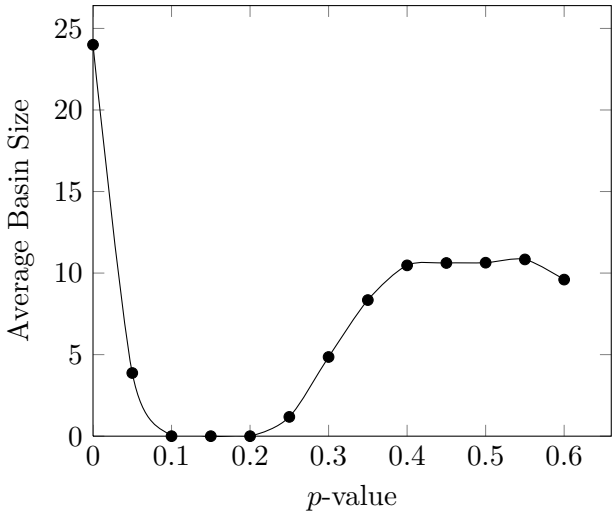


Figure 3.4: Average Basin size of patterns belonging to one cluster generated using T1

Comments

The results are displayed in figure 3.4. This initial experiment is confirming not only what Federico obtained [8], but also what we already know from theory: A Hopfield network that is trained with similar patterns loses capacity. Although we are not measuring capacity here, the basin size is strongly correlated to that.

Initially, when p is zero, we observe a huge value of 24 for the basin size. This is easily explained, since all the patterns generated are the the same as the root pattern. Subsequently, the network is trained with the same patterns that have the effect of creating a super-attractor with a huge basin size.

When p is between 0 and a critical value of 0.35, the basin size is 0, since the patterns are too close to each other and don't have enough space to fit basins of attraction between them. As soon as p goes over 0.35, the basins start to increase until a maximum size of 11.

Further explanation of the results

The results can also be easily understood and visualised in figure 3.5. For small p values, the attractors formed in the network are too close to each other to allow room for any basins of attraction (apart from the attractors lying at the edge of the cluster). When the p-value is high, the attractors become more disperse, leaving room for larger basins. It is also worth mentioning that the basin shapes are not necessarily hyper-spherical.

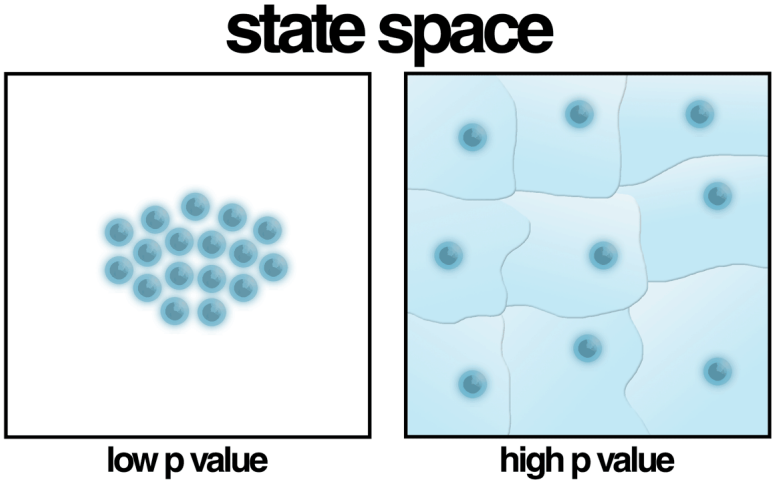


Figure 3.5: Illustration of clusters of attractors generated using T1, with different p-values.

3.5.2 Basin Size for Two Clusters using T1

This experiment is similar to the previous one, however it contains two clusters this time. The value p_1 for one cluster will stay the same (fixed at 0.45), while p_2, corresponding to the second cluster, will vary in the range [0-0.5]. Method T1 is used, for a Hopfield Network of N neurons. The procedure is given below:

I. We generate 2 random patterns μ_1 and μ_2, corresponding to clusters C_1 and C_2.

II. We generate P patterns for C_1, using T1 with associated probability $p_1 = 0.45$.

III. For all values of probability p_2 from 0 to 0.5

1. Starting from μ_2 we generate P patterns using T1 with associated probability p_2, and give them to the Hopfield network in order to be learned according to Hebb or Storkey rule.
2. For both sets of patterns, we measure the mean basin size using the Storkey-Valabregue measurement and plot the values on the graph.

We will be interested to observe how can the probability p influences the basin sizes. For this reason, we have run two experiments, in which we are outputting the basin sizes for cluster 1, in which p varies. The

Comments

This latter experiment seems to suggest that the probability p of flipping bits in the pattern doesn't actually inflence the basin sizes of attractors in Cluster 1. The same behaviour has been observed in the previous graph, and this is the case especially when the root images, that are randomly generated, are far away from each other.

Link to attachment theory

A network having been trained with two clusters of patterns can be interpreted, in attachment theory, with an infant that has been exposed to two different caregivers. The patterns can be for example, visual or auditory memories describing the 2 caregivers.

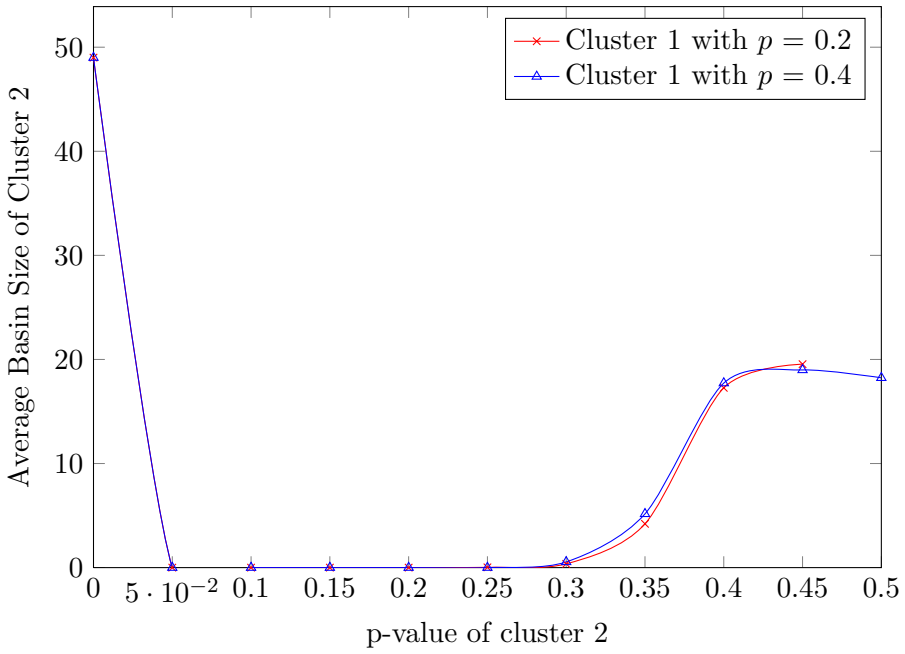


Figure 3.6: Average Basin size of patterns belonging to Cluster 2, generated using T1

3.6 Experiments with Gaussian-distributed patterns

In this section we are testing clusters of patterns that have been generated using a normal distribution. We are expecting to get similar results to the T1 method, since by the Central Limit Theorem, the Binomial distribution B(N, p) is nicely approximated by a Gaussian distribution with mean N(Np, Np(1-p)). Since in the previous experiments, we used to increase the probability p of flipping a bit, this now translates to increasing the standard deviation of the normal distribution.

Comments

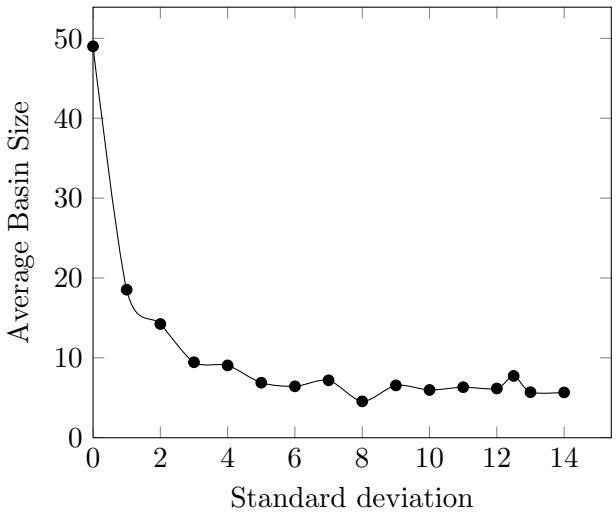


Figure 3.7: Average basin sizes of one cluster generated using T2

The results here are quite surprising. The experiment with Gaussian-distributed patterns shows us that the basin sizes decrease exponentially as standard deviation increases. This might

be the case because at low standard deviation, the patterns would repeat themselves and create small super-attractors that have big basin sizes. This is in contradiction with what we obtained previously, when using method T1, and further details and possible explanations are given in section 6.2.1.

3.6.1 Basin Size for Two Clusters using T2

Comments

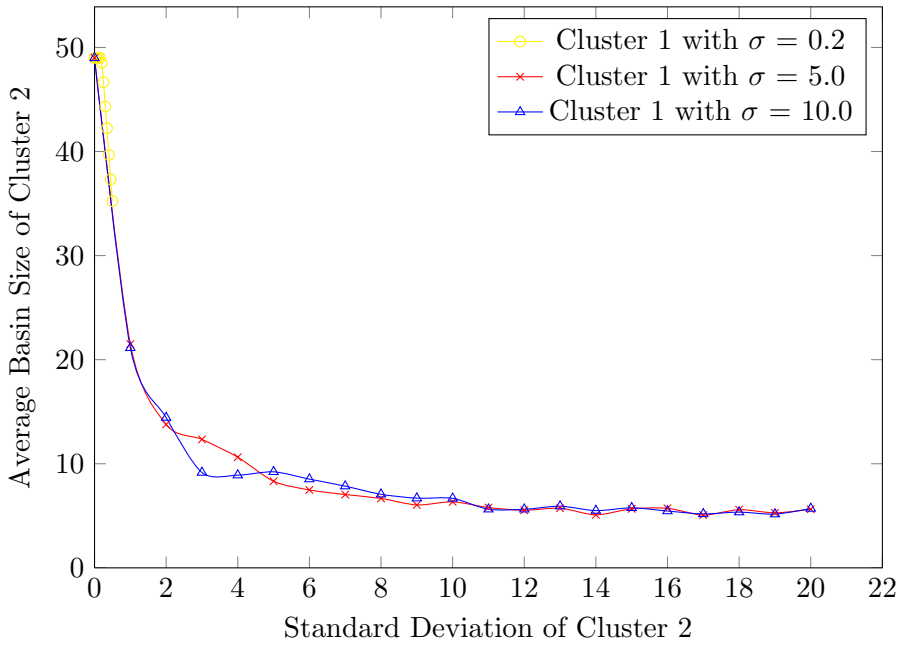


Figure 3.8: Average basin size for two clusters generated using T2

This experiment has been run on a network of 100 nodes, with 2 clusters of patterns:C1, with fixed standard deviation σ_1, and C2, with varying standard deviation σ_2. We have run 3 diffeent trials, with $\sigma_1 = 0.2$, 5 or 10

As it can be easily noticed in the graph, the standard deviation does not have any effect on the basin size of the clusters. This seems to suggest that a more disperse cluster will not be affected by neighbouring attractors.

3.7 Super Attractors

In the context of learning models such as the Hopfield model, we define a super attractor as an attractor resulting from training a model with multiple occurrences or instances of some stored pattern. The degree of a super attractor denotes the number of occurrences of its corresponding pattern in the training set.

In the context of modelling Attachment theory, a super attractor may represent repeated interactions with the primary care giver. Clearly, it is of interest to investigate the properties of such an attractor; in particular establishing the existence and, if existent, the type of relationship between the degree of the super attractor and the extent of its dominance, or stability, over the space of patterns.

3.7.1 Single super attractor

1. Fix N, the number of neurons.

2. Choose a random pattern p_{super}, which signifies the primary care giver.

3. Choose a number of random patterns $\overrightarrow{p}_{random}$, such that the Hamming distance between $p_{super}$and each of $\overrightarrow{p}_{random}$is between 25% and 75%.

   The range forms a ball centred at 50% Hamming distance[1] with an arbitrary radius, chosen such that the probability of a $\overrightarrow{p}_{random}$falling into p_{super}'s basin of attraction is small. This is done to avoid forming clusters of attractors, which we deal with separately in Recall that the Hopfield network is sign blind, and as a result the inverse of p_{super}, p_{super}^{-1}, forms a symmetric super attractor. It is for this reason that a symmetric range about 50% is chosen.

4. Choose a degree d for $p_{super}$and train a Hopfield network using $\overrightarrow{p}_{super}^{d}$ (d instances of p_{super}) and $\overrightarrow{p}_{random}$.

5. Measure the basin of attraction of p_{super}using the Storkey-Valabregue method.

6. Repeat from (2) for various values of degree d.

In our experiment 100 neurons are used, and the network is trained with 16 random patterns in addition to the super attractor. It is run with degree values [1, 2, 4, 8, 16, 32]. The entire procedure is repeated 800 times for various randomly chosen $p_{super}$and $\overrightarrow{p}_{random}$. The average results obtained are summarised in 3.9.

This experiment can be replicated by running `Experiment.hs`.

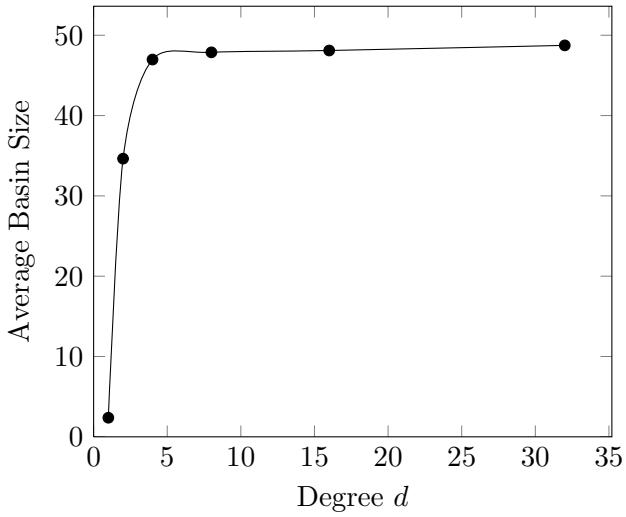


Figure 3.9: Average basin of attraction for a super attractor with varying degrees.

The results show that as the super attractor's degree is increased, its basin of attraction also increases. Also note that this increase appears to approach the singularity at 50% Hamming distance, which is consistent with our knowledge of p_{super}'s symmetrical counterpart. The very fast initial growth also indicates that super attractors in general very stable and demonstrate strong dominance over the space of patterns.

Linking back to the Attachment theory model, this may be interpreted as depicting the strong influence of repeated and consistent interaction with the child, represented by a super attractor of increasing degree. In particular, it exhibits its dominance over distantly scattered, unrelated influences, which are represented by the random patterns.

[1]The percentage Hamming distance is simply the Hamming distance divided by the number of bits N

3.7.2 Two super attractors

1. Fix N, the number of neurons.
2. Fix a degree d_{origin}, for all chosen $p_{origin}$from (3).
3. Choose a random pattern p_{origin}, which signifies the primary care giver.
4. Choose a random pattern p_{new}, such that the Hamming distance between $p_{origin}$and each of $p_{new}$is between 25% and 75%. This could symbolize the training due to either an alternate and influential care giver, or that resulting from a retraining period.
5. Choose a number of random patterns $\overrightarrow{p}_{random}$, such that the Hamming distance between $p_{origin}$and each of $\overrightarrow{p}_{random}$is between 25% and 75%.

   Note that while this allows for the possibility of forming a cluster in the vicinity of p_{new}, in practice the probability of this occurring is negligible.
6. Choose a degree $d_{new}$for $p_{new}$and train a Hopfield network using $\overrightarrow{p}_{random}$, d_{origin}instances of p_{origin}, and d_{new}instances of p_{new}
7. Measure the basins of attraction for each of $p_{origin}$and p_{new}using the Storkey-Valabregue method.
8. Repeat from (3) for different value of degree d_{new}.

In our experiment 100 neurons are used, and the network was trained with 8 random patterns in addition to the original and new super attractors, $p_{origin}$and p_{new}, respectively. $p_{origin}$is given a fixed degree, d_{origin}, of 8. This value was chosen as being the smallest point of stability from the previous experiment 3.9, beyond which increasing the degree further does not greatly impact the basin of attraction. It is run with d_{new}degree values [1, 2, 4, 8, 16, 32]. The entire procedure is repeated 634 times for various randomly chosen p_{origin}, $p_{new}$and $\overrightarrow{p}_{random}$. The average results obtained are summarised in 3.10.

This experiment can be replicated by running `ExperimentSuper2.hs`.

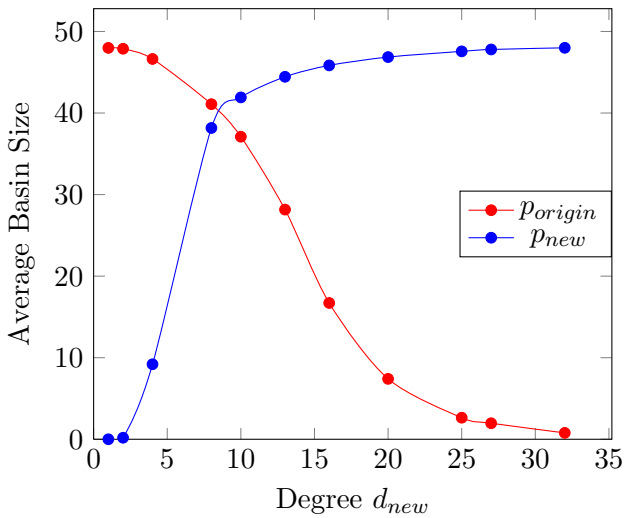


Figure 3.10: Average basin of attraction for the two super attractors, p_{origin}having a fixed degree, d_{origin}, of 8, and varying the degree, d_{new}, of p_{new}.

The results clearly reveal that increasing the degree of the new pattern increases its own basin of attraction, and decreases that of the original pattern. This relationship is symmetrical,

as can be observed by the shape of the graph, in addition to the intersection point, which occurs (roughly) near the point (8 , 8) where both patterns have the same degree of 8. By random choice of original pattern, and random choice of new pattern (subject to a Hamming distance of 25% to 75% from the original pattern), we can claim that the basin of attraction depends solely on the relative degrees of the two patterns, and is independent of the actual patterns themselves. This is also consistent with the symmetry observed.

This also fits in nicely nicely with in the Attachment theory interpretation. The original pattern as before represents the consistent interactions with a primary care giver. The new pattern may represent the consistent, yet significantly different, interactions with an alternate and influential care giver, perhaps the other parent of the child or a relative with whom they have developed a close relationship. Intuitively, the care giver which has more interactions would exert a stronger influence, undermining the dominance of the other care giver, and indeed the impact of any other interactions.

In relation to retraining, the new pattern may also be taken to represent the influences arising due to the undertaken retraining phase. We find that, similar to the above, increasing the number of interactions due to retraining further ingrains its dominance, and dilutes that of the previous interactions due to the primary care giver or otherwise.

3.8 Comparing Storkey and Hebbian learning

When training a Hopfield network, we have employed two types of learning: the first one is the Hebbian learning, while the second one is Storkey learning. The advantage of the latter is that it increases the capacity of the network and the basin of attraction of the clusters. We will now explain some experiments we did with both types of learning in order to see how the learning type affects the basin of attraction for clusters.

We will now show how the learning determines the basin of attraction of a Gaussian distributed cluster. The generation was done using the T2 method described in section 3.5.

Learning	N	Cluster size	μ	σ	Average size of basin of attraction
Hebbian	50	2	25	5	10.0
Storkey	50	2	25	5	18.0
Hebbian	50	4	25	5	1.5
Storkey	50	4	25	5	4
Hebbian	50	4	25	10	5.75
Storkey	50	4	25	10	8.0
Hebbian	50	5	25	5	4.4
Storkey	50	5	25	5	4.8
Hebbian	50	6	25	5	4.2
Storkey	50	6	25	5	4.4
Hebbian	50	6	25	10	4.45
Storkey	50	6	25	10	5.61

Table 3.1: Results comparing Storkey and Hebbian learning for various parameters

The results confirm what the mathematical theory showed us: Storkey learning increases the average basin of attraction. We must note that this does not come without a price: training the network using Storkey learning slowed down our experiments, as it is more expensive. Depending to the application, this is an acceptable trade off. All our functions and experiments can be performed with both types of learning, just by changing a parameter (the learning type), which enables any user of our libraries to make the choice depending on the use case.

Chapter 4

Design and Implementation

4.1 Technologies used

Haskell

We have combined several technologies in order to create a nice workflow for our working environment. First of all, it is worth mentioning that we implemented the core algorithms for the neural network in Haskell. It is a very good candidate for solving mathematical problems, and has a strong type system that helped us detect bugs early on, at compile time, and made refactorings easy.

We also evaluated Python and Scala as our main implementation language for this project; Haskell was eventually chosen because of its strong, static type system (compared to Python), its easy integration with C (compared to Scala), and the personal learning interests of our team.

Image loading with C

The functions responsible for image preprocessing (loading, rescaling, converting to gray-scale values, mapping pixels to binary patterns) were implemented in the **C** programming language, using the MagickWand ibrary which is part of ImageMagick. We used Haskell's *Foreign Function Interface* for the communication between C and Haskell.

Git

Since this was a group project in which all of us has to implement various parts of the system, we used Git as the version-control system that can keep track of our files, automatically merge source code and backup our data.

Test-driven development with QuickCheck

Although Haskell as a programming language is already quite good at avoiding programming errors, it is not a theorem prover. In our effort to keep our code correct and find errors early, we therefore maintained an extensive test suite structured with **Hspec**, where we performed both unit tests on lower level implementations and as well as behavioural tests testing the functionality of the full program. We barely had to write tests in the example-driven style often found in imparative and dynamically typed languages, as we used **QuickCheck** to express the required functionality as lambda functions in first order logic. QuickCheck then generates thousands of tests cases automatically, based on the types our functions take as input, and also shrinks failing test cases to the smallest input that still fails. Using this technique, we found many failing edge cases in our initial implementation, and managed to keep the code size of our test suite small because most functions could be covered using only one or two first-order predicates.

Code Reviews and Continuous Integration

With the third year project being the largest group effort in our course, we tried to set ourselves new standards in using tools and techniques that help or benefit from the team. We set up an instance of the **ReviewBoard** web-based code review review system and enforced that all commits to our code base were reviewed by other team members before integration in the stable branch. We could avoid a lot of bugs with this and it also had the positive effect that the whole group agreed on the style and conceptual structure of the code. We also set up our own **Jenkins** Continuous Integration server on a lab machine that pulled the code, built it and ran the test suite as soon as a change had been made; using this, we could get rapid feedback on new changes and whether they broke the build or introduced a correctness or performance regression. The code always being in a buildable state after every single commit also saved us from wasting time to manually find out afterwards what change broke it.

4.2 Recognition System

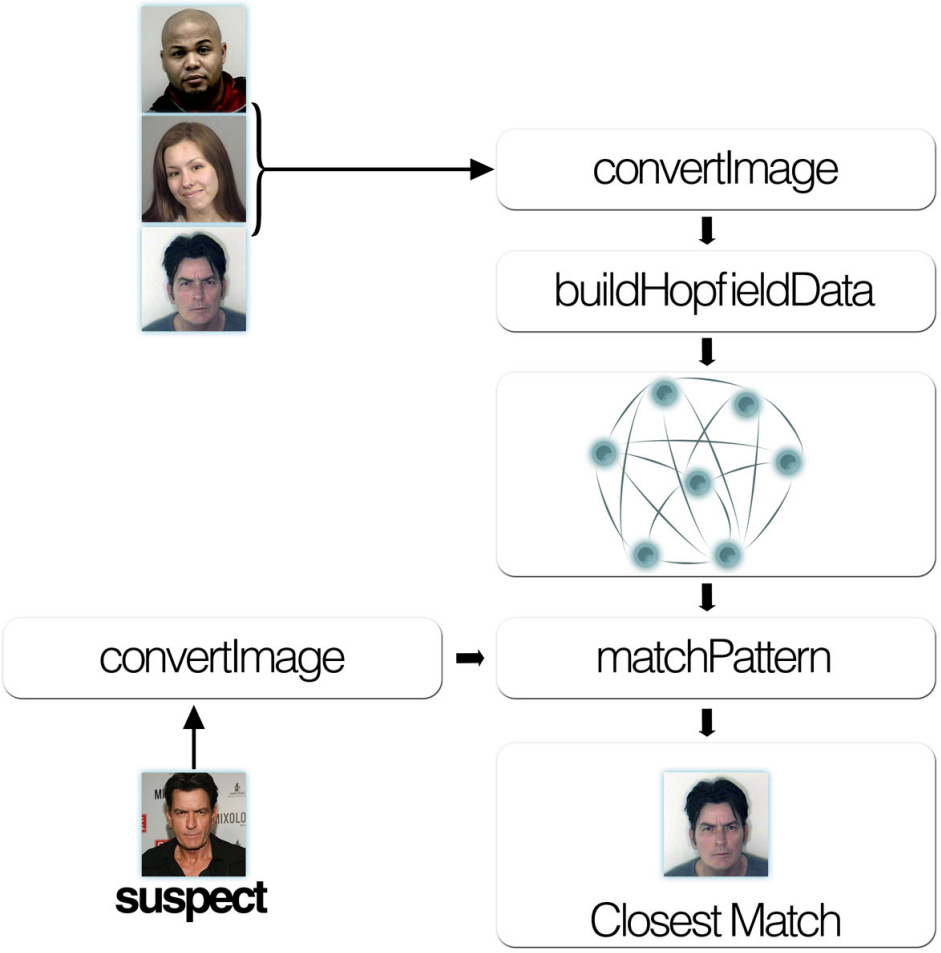


Figure 4.1: Outline of the Recognition System demonstrating how the various system components interact

4.3 Technical Challenges

4.3.1 Computation Time

The common theme of our experiments is measuring and comparing basins of attractions over a multitude of parameters, performed using the Storkey-Valabregue method (see 3.3.1). As we know, this involves performing a very expensive computation: sampling 100 patterns for each Hamming radius, a maximum of N steps, and for each of those patterns iterative updates are run with the network until convergence is achieved. Though impossible due to properties of Hopfield networks, we can approximate the worst case complexity of the latter as requiring 2^N iterations until convergence, 2^N being the number of possible states. Thus we may cynically, for the sake of quantifying it, approximate the worst case complexity of the Storkey-Valabregue method to be $O(N2^N)$. While this is probably not an accurate representation of what occurs in practice, it serves to demonstrate the sheer amount of computation performed.

Naturally, this posed a significant computational challenge which we had to be overcome, as we typically obtain a large number of samples of basin measurements for our experiments, in the order of hundreds of samples for a given experiment. One way in which we overcome this limitation is by exploiting parallelism both at the process level (trivially achieved using **Haskell**!) and at the machine level, running multiple experiments on several lab machines. We also sought to improve the quality of our code itself, using profiling to identify hotspots.

4.3.2 Big Data

Another challenge which we have faced is dealing with the sheer amount of data generated by our experiments. The data typically needs to be collected, processed, and presented in a consistent manner. In order to address this issue, we attempted to automate such tasks. Automation is key, as it saves time, is not prone to (low level) human error, and generally scales well. Achieving this goal required the joint cooperation of three areas:

- Experiment executables should output data in a sensible and easy to parse format. Ideally, the output data should not be heavily processed so as to allow flexible use of it.
- Writing scripts which format and aggregate data, typically calculating the mean value.
- Presentation of data in the form of graphs of tables ought to be consistent. Ideally, the data should be kept separate from the format or design of its final presentation form. The `pgfplots` package for LaTeXwas very helpful for this.

4.4 Anticipated Risks

4.4.1 Failure to match a pattern stored

One of the risks associated with using the Hopfield model is that the converged pattern may not be one of the stored patterns, but rather a spurious pattern, as explained in section 2.3.3. This poses a problem for our recognition application, which needs to return as a result the closest matched stored pattern. Another issue with similar repercussions is that even after a large number of iterations are run, the pattern has not yet converged. One solution to this is to simply output the closest stored pattern available. More precisely, the pattern that has the smallest Hamming distance between it and the current state of the system. This makes our image recognition robust to such unfortunate occurrences.

4.4.2 A stored pattern not a fixed point

It is possible that the given set of training patterns are such that some of the patterns simply cannot be "stored" in the network, i.e. it is not a fixed point of the network. The probability of this occurring increases as the relative number of training patterns with respect to the number of neurons increases. The best way to deal with this problem is to perform a check (using `checkFixed`) and, if detected, report it to the user and allow them to take action. The typical remedy for this would be to increase the number of neurons in the network, at the expense of increased resource usage in both space and time.

4.4.3 Floating point calculation

As we are dealing with extensive calculations of floating point value, we must be careful when dealing with such quantities. In particular, equality comparisons are likely to yield unexpected results. This has had an impact on some tests as well as functions such as `validWeights`. We have taken care of this problem by replacing equality comparisons and similar with fuzzy equality comparisons, with some accepted margin of error ϵ.

Chapter 5

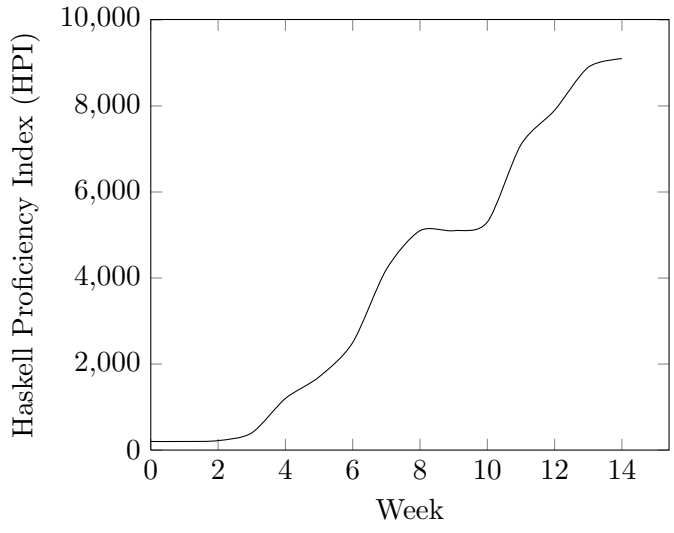
Evaluation

5.1 Implementation

In a nutshell, we are content with the choice of our tools and with our implementation of the neural network. Since our mathematical core of the algorithm does not contain much state information, we chose to implement our project in Haskell. The strong types allowed us to have confidence in the code we write. Since most of our experiments took a lot of computation time, we had to make sure they run seamlessly and don't crash in the middle of the execution because of a pattern match failure.

There are a lot of advantages in using Haskell, including its clarity and correctness. The type system enforces the programmer to have a clear understanding of the problem and the various solutions available, which leads to high quality code. Using Haskell came with an initial price: while all the group members were familiar with the language due to our course in first year, we soon realised that we only played with Haskell before. Because of our project, we now deeply appreciate its true beauty and power.

This early overhead turned out to be very small compared to the benefits and the satisfactions we got back. The project enabled us to learn about neural networks and attachment types, but we chose to exploit this opportunity in order to become better using a programming language we like (see fig 5.1).



1

Figure 5.1: Average Haskell Proficiency Index over time.

5.1.1 Optimising for speed

While testing the recognition application, we realised that our implementation was not as fast as we would have liked it to be, and imposed serious limitation on the size of the networks we were able to try out. As a consequence, we invested a good amount of time in understanding Haskell compiler optimisations, runtime characteristics, and profiling tools. Using time and allocation profiling, we found that two inner loops in our numerical network updating code were slowing us down significantly. By changing 12 lines to manually unroll the involved list comprehensions into tail-recursive functions we were able to get an overall 4-times speedup of our program. Additionally, Haskell's being a purely functional programming language allows for easy and efficient parallelism. By replacing a top-level map with a parallel map function, our program immediately scaled up to the eight processes that were available on the lab machines we used. Because our main processing time is spent in numerical, cache-local loops, we also got almost 8-times improved performance from this. This final scaling success is an improvement over an initial try in which we parallelized an inner loop, which turned out to be too fine-grained.

5.1.2 Unit Testing

Unit testing of our Haskell functions was performed using the HUnit framework. This allowed us to heavily test the complex functions responsible for performing updates or calculating energy and capacity, thus making sure that important invariants always hold: energy is always monotonically decreasing, capacity is following the theoretical trends and updates eventually converge to an attractor.

5.2 Evaluating our deliverables

Our main task was twofold:

1. Implement the Hopfield model and a basic image recognition application to show its functionality.

2. Support or reject Federico's claims about clustering and evaluate his results in the context of Attachment Types.

We implemented both of these goals: We have a clear, concise Haskell implementation of the Hopfield model with a public and well-tested API to train Hopfield networks and converge patterns in them. To show that it works in the real world, we built a police-style suspect face matching application with an accessible GUI, which in addition to performing image recognition also has a functionality to introduce a layman to how Hopfield networks work using interactive examples.

We implemented Federico's clustering experiments and checked the majority of his claims; as described in the analysis above, our independent results largely support his claims.

Extending our initial goals, we additionally evaluated the influence of Super Attractors and their relation to Attachment Theory; we can conclude that Federico's results concerning clustering and our results concerning Super Attractors support the idea that Hopfield networks are a good way to model the real world domain of Attachment Types.

[BOLZMANN]

Finally, we came up with some our own way to use the Boltzmann machine for pattern matching, implemented the Storkey method as an alternative to Hebbian learning for increased network capacity, and have proven various properties about it, see Appendix 1 for more details.

Chapter 6

Conclusion and Future Extensions

6.1 Analogy with Attachment Theory

The thorough analysis of the attractor neural networks (Hopfield network and Boltzmann machine) has clearly proven their capabilities, being able to model complex attachment types. However, research done in linking the model with the real-life scenario is quite slim, and further work needs to be dedicated especially in this area.

6.2 Future Extensions

We have introduced the reader with only a few flavours of what Attractor neural networks can really do. Deep investigations should be done in order to further investigate their properties, and really push them to the limit. These might provide a good source of inspiration for developing even more biologically realistic models, that better simulate the human brain.

Another line of extension is about providing better visualisation tools for these highly abstract models, that work on N-dimensional spaces. Parallel coordinates can be used for easily visualizing the N-dimensional attractors, or the convergence of the network to such an attractor. Furthermore, plots can be done on the available number of neurons that can be updated at each step in the convergence process.

The experiments we did on Hopfield networks should be reproduced using the Restricted Boltzmann machine, which is a more powerful neural network. Furthermore, biological plausibility also needs to be kept in mind, which would surely provide us with more efficient learning systems.

6.2.1 Inconsistencies in the results

It is important mentioning a few apparent inconsistencies in the cluster experiments. For examples, the experiments that have been done, using 1 cluster, display inconsistencies in the results between methods T1 and T2. We believe the main reason for this inconsistency is the limitation of the methodology for sampling Gaussian-distributed patterns and the poor use of the state space. Another reason for this apparent inconsistency might also be the presence of spurious attractors, that can interfere with the real attractors and "steal" part of their basins.

Another reason for the inconsistencies might also be the chaotic shape of the basins. Since the Storkey-Valabregue measurement is only good for hyper-spherical basin shapes. A more powerful method, such as a Monte Carlo sampling in the neighbourhood, is necessary for measuring basins that don't have hyperspherical shapes.

Chapter 7

Project Management

7.1 Project Organisation

We started out the project by thoroughly investigating the Hopfield model, reading important research papers that were describing it in detail and showing the potential for other applications.

Once we had a firm understanding of the mathematical model, we assigned various tasks to each members in order to create a simple image recognition application. Wael and Mihaela started implementing the core algorithm in Haskell, based on information gathered from the research papers. Razvan implemented the image preprocessing in C, while Niklas was integrating the Haskell algorithmic core with the C programs. Lukasz worked on implementing the GUI and image loading, GUI usability and correctness testing, as well as supplying much of the artistic work, including that found throughout this report.

We heavily followed the advice that Dr. Chatley offered in the Software Engineering Lectures. Therefore, we used to have regular weekly meetings and perform iterative development in order to make sure we have a working version of our product at any time.

Although we did not use any popular software engineering methodology, such as Agile development, we did make sure we are on track with the project requirements. We closely collaborated in order to ensure timely delivery and were flexible to quick changes that came across the progress of the project.

7.2 A story

It is in difficult times that the value of team work really flourishes, as the story of our experiments shall show. As aforementioned, running our experiments posed as a formidable difficulty due the hefty processing time it required. Such experiments would last long hours, sometimes overnight. This was deemed unacceptable to us, and we sought a solution to this problem. Various members of the team collaborated in subgroups, each focusing on a particular aspect of improvement. These sub-groups were not rigid subdivisions of the team, quite the contrary, they were dynamic and self-arranging. One task undertaken by a subgroup is the optimisation of the experiment, in particular those arising from running the Storkey-Valabregue method. Iteratively optimising the code and testing the results locally and at scale (by actually running the experiments). Another subgroup worked on exploiting the resources available to us, developing the code and writing scripts to parallelize the experiments to run on multiple cores and multiple machines. Eventually, the experiments exhibited an improvement of at least 30 fold for some experiments (the magic of parallelism!). So what can we conclude from this? Team work can perform miracles!

Chapter 8

Bibliography

[1] G.E. Dahl, D. Yu, L. Deng, and A. Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):30–42, 2012.

[2] Abbas Edalat. Hopfield networks. Part of Complex Systems course at Imperial College London, 2012.

[3] D. Gil and D.J. Manuel. Diagnosing parkinson by using artificial neural networks and support vector machines. *Global Journal of Computer Science and Technology*, 9(4), 2009.

[4] G. Hinton. A practical guide to training restricted boltzmann machines. *Momentum*, 9:1, 2010.

[5] Alice Julien Lafferiere. Hopfield network.

[6] H. Larochelle and Y. Bengio. Classification using discriminative restricted boltzmann machines. In *Proceedings of the 25th international conference on Machine learning*, pages 536–543. ACM, 2008.

[7] J. Louradour and H. Larochelle. Classification of sets using restricted boltzmann machines. *arXiv preprint arXiv:1103.4896*, 2011.

[8] Federico Mancinelli. Modelling attachment types with hopfield neural network. Master Thesys for Computing, Imperial College, 2012.

[9] V. Nair and G.E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proc. 27th International Conference on Machine Learning*, pages 807–814. Omnipress Madison, WI, 2010.

[10] A. Storkey. Increasing the capacity of a hopfield network without sacrificing functionality. *Artificial Neural NetworksICANN'97*, pages 451–456, 1997.

[11] A.J. Storkey and R. Valabregue. The basins of attraction of a new Hopfield learning rule. *Neural Networks*, 12(6):869–876, 1999.

[12] Y.W. Teh and G.E. Hinton. Rate-coded restricted boltzmann machines for face recognition. *Advances in neural information processing systems*, pages 908–914, 2001.

[13] R.S. Wedemann, R. Donangelo, and L.A.V. de Carvalho. Generalized memory associativity in a network model for the neuroses. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 19(1):015116–015116, 2009.

[14] Wikipedia. Attachment patterns.

[15] Wikipedia. Attachment theory.

[16] Wikipedia. Hebbian theory.

[17] Wikipedia. Hopfield network.

Appendix A

Errors in Hopfield networks

When a Hopfield network is trained using a list of patterns, it is desired that those patterns are attractors, which requires that they are fixed points. In case one of the patterns is not a fixed point, it is said that the network has an error.

We will now derive the equations for errors in Hopfield networks. Firstly, we shall derive the probability of error for a pattern which was used for training, given that all the patterns were independent. Secondly, we will derive the equation of error for a super attractor: a pattern which was used to train the network, but which appeared in the training multiple times. The assumption for this derivation is that the other training patterns are independent. In the context of attachment types, we can assume that the super attractor is the primary care giver, which was consistent in his/her behaviour, and that the other encounters of the child were independent.

A.1 Stability of stored states, given independence of all stored patterns

Consider a pattern stored in the network x^k. We want to find out the probability of error for that pattern: how probable is that given that x^k is stored in the network it is not a fixed point?

Given the update formulae for neuron i in pattern k (1) and the formulae for the weights, according to the training of the network (2):

$$h_i^k = \sum_{j=1}^{N} w_{ij} x_j^k$$

$$w_{\mathrm{ij}} = \frac{1}{N} \sum_{l=1}^{p} x_i^l x_j^l \tag{A.1}$$

we obtain by substitution:

$$h_i^k = \sum_{j=1}^{N} \left(\frac{1}{N} \sum_{l=1}^{p} x_i^l x_j^l \right) \cdot x_j^k = \frac{1}{N} \sum_{j=1}^{N} \sum_{l=1}^{p} x_i^l x_j^l x_j^k =$$

$$= \frac{1}{N} \sum_{j=1}^{N} x_i^k x_j^k x_j^k + \frac{1}{N} \sum_{j=1}^{N} \sum_{l=1, l\neq k}^{p} x_i^l x_j^l x_j^k$$

$$= \frac{1}{N} \sum_{j=1}^{N} x_i^k + \frac{1}{N} \sum_{j=1}^{N} \sum_{l=1, l\neq k}^{p} x_i^l x_j^l x_j^k = x_i^k + \frac{1}{N} \sum_{j=1}^{N} \sum_{l=1, l\neq k}^{p} x_i^l x_j^l x_j^k$$

The new value of neuron i $x_i^{k'}$ is given by the sign of h_i^k, so:

$$h_i^k x_i^k \begin{cases} \geqslant 0 \text{ if } x_{i'}^k = x_i^k \\ < 0 \text{ otherwise} \end{cases} \Rightarrow -h_i^k x_i^k \begin{cases} \leqslant 0 \text{ if } x_{i'}^k = x_i^k \\ > 0 \text{ otherwise} \end{cases} \tag{A.2}$$

$$-h_i^k x_i^k = -x_i^k x_i^k - x_i^k \cdot \left(\frac{1}{N} \sum_{j=1}^{N} \sum_{l=1, l\neq k}^{p} x_i^l x_j^l x_j^k \right) = 1 \underbrace{- \frac{1}{N} \sum_{j=1}^{N} \sum_{l=1, l\neq k}^{p} x_i^l x_j^l x_j^k x_i^k}_{C_i^k} \tag{A.3}$$

From (3) and (4) we conclude that we get an error if $C_i^k > 1$, so the probability of getting an error is the given by the probability of $C_i^k > 1$.

We model C_i^k as follows. Each $x_i^l x_j^l x_j^k x^k$ can be modelled as a sample drawn from a random variable X with E(X) = 0 and $\mathrm{Var}(X) = E(x^2) - E(x)^2 = 1.0$

By using the central limit theorem and by approximating $N\,(p$ - $1)$ to Np :

$$\frac{1}{Np} \sum_{i=1}^{Np} X_i \sim N\left(0, \frac{1}{Np}\right)$$

Thus,

$$\frac{1}{N} \sum_{i=1}^{Np} X_i \sim N\left(0, \frac{p}{N}\right)$$

$$C_i^k \sim N\left(0, \frac{p}{N}\right)$$

By denoting $\frac{p}{N}$ with σ :

$$P\left(C_i^k > 1\right) \simeq \frac{1}{\sqrt{2\pi}\sigma} \int_1^{\infty} e^{-\frac{x^2}{2\sigma^2}}\,\mathrm{dx} = \frac{1}{\sqrt{2\pi}\sigma} \int_{\frac{1}{\sqrt{2}\sigma}}^{\infty} e^{-y^2} \sqrt{2}\sigma\,\mathrm{dy} =$$

$$= \frac{1}{\sqrt{\pi}} \int_{\frac{1}{\sqrt{2}\sigma}}^{\infty} e^{-y^2}\,\mathrm{dy} = \frac{1}{\sqrt{\pi}} \left(\int_0^{\infty} e^{-y^2}\,\mathrm{dy} - \int_0^{\frac{1}{\sqrt{2}\sigma}} e^{-y^2}\,\mathrm{dy} \right) =$$

$$= \frac{1}{\sqrt{\pi}} \int_0^{\infty} e^{-y^2}\,\mathrm{dy} - \frac{1}{\sqrt{\pi}} \int_0^{\frac{1}{\sqrt{2}\sigma}} e^{-y^2}\,\mathrm{dy} = \frac{1}{\sqrt{\pi}} * \frac{\sqrt{\pi}}{2} - \frac{1}{2} \operatorname{erf}\left(\frac{1}{\sqrt{2}\sigma}\right) =$$

$$= \frac{1}{2}\left(1 - \operatorname{erf}\left(\frac{1}{\sqrt{2}\sigma}\right)\right) = \frac{1}{2}\left(1 - \operatorname{erf}\left(\sqrt{\frac{N}{2p}}\right)\right)$$

$$\text{where } \operatorname{err} x = \frac{1}{\sqrt{2\pi}} \int_0^x e^{-x^2}\,\mathrm{dx}$$

A.2 Stability of a super attractor, given independence of all other stored patterns

By following the above way of reasoning, we compute the probability of error for a super attractor.

Given that the super attractor has degree d (it appears d times in the list of patterns used to train the network), we compute the probability that it is not a fixed point.

Let x^k be the supper attractor with degree d.

Let S = $\{i \mid x^i = x^k$, i $\in \{1 \dots p\}\}$, be the set of indexes of occurences of the supper attractor x^k.

We assumed that the super attractor has degree $d \Rightarrow |S| = d$.

$$h_i^k = \sum_{j=1}^{N} \left(\frac{1}{N} \sum_{l=1}^{p} x_i^l x_j^l \right) \cdot x_j^k = \frac{1}{N} \sum_{j=1}^{N} \sum_{l=1}^{p} x_i^l x_j^l x_j^k =$$

$$\frac{1}{N} \sum_{j=1}^{N} \sum_{l \in S}^{p} x_i^k x_j^k x_j^l + \frac{1}{N} \sum_{j=1}^{N} \sum_{l \notin S}^{p} x_i^l x_j^l x_j^k = \frac{1}{N} \sum_{j=1}^{N} \sum_{l \in S}^{p} x_i^k x_j^k x_j^k + \frac{1}{N} \sum_{j=1}^{N} \sum_{l \notin S}^{p} x_i^l x_j^l x_j^k =$$

$$\frac{1}{N} \sum_{j=1}^{N} \sum_{l \in S}^{p} x_i^k + \frac{1}{N} \sum_{j=1}^{N} \sum_{l \notin S}^{p} x_i^l x_j^l x_j^k = \frac{1}{N} N d x_i^k + \frac{1}{N} \sum_{j=1}^{N} \sum_{l \notin S}^{p} x_i^l x_j^l x_j^k =$$

$$= d x_i^k + \frac{1}{N} \sum_{j=1}^{N} \sum_{l \notin S}^{p} x_i^l x_j^l x_j^k$$

Thus

$$-h_i^k x_i^k = -d x_i^k x_i^k - \frac{1}{N} \sum_{j=1}^{N} \sum_{l \notin S}^{p} x_i^l x_j^l x_j^k x_i^k =$$

$$-d - \underbrace{\frac{1}{N} \sum_{j=1}^{N} \sum_{l \notin S}^{p} x_i^l x_j^l x_j^k x_i^k}_{S_i^k}$$

In order to get an error, minus $h_i^k x_i^k \leqslant 0$, so $S_i^k \geqslant d$. By using the same reasoning as before,

$$\frac{1}{N(p-d)} \sum_{i=1}^{N(p-d)} X_i \sim N\left(0, \frac{1}{N(p-d)}\right)$$

$$\frac{1}{N} \sum_{i=1}^{N(p-d)} X_i \sim N\left(0, \frac{p-d}{N}\right)$$

$$S_i^k \sim N\left(0, \frac{p-d}{N}\right)$$

By denoting $\frac{p-d}{N}$ with σ :

$$P\left(S_i^k > d\right) \simeq \frac{1}{\sqrt{2\pi}\sigma} \int_d^{\infty} e^{-\frac{x^2}{2\sigma^2}} \, \mathrm{dx} = \frac{1}{\sqrt{2\pi}\sigma} \int_{\frac{d}{\sqrt{2}\sigma}}^{\infty} e^{-y^2} \sqrt{2}\sigma \, \mathrm{dy} =$$

$$= \frac{1}{\sqrt{\pi}} \int_{\frac{d}{\sqrt{2}\sigma}}^{\infty} e^{-y^2} \, \mathrm{dy} = \frac{1}{\sqrt{\pi}} \left(\int_0^{\infty} e^{-y^2} \, \mathrm{dy} - \int_0^{\frac{d}{\sqrt{2}\sigma}} e^{-y^2} \, \mathrm{dy} \right) =$$

$$= \frac{1}{\sqrt{\pi}} \int_0^{\infty} e^{-y^2} \, \mathrm{dy} - \frac{1}{\sqrt{\pi}} \int_0^{\frac{d}{\sqrt{2}\sigma}} e^{-y^2} \, \mathrm{dy} = \frac{1}{\sqrt{\pi}} \frac{\sqrt{\pi}}{2} - \frac{1}{2} \operatorname{erf}\left(\frac{d}{\sqrt{2}\sigma} \right) =$$

$$= \frac{1}{2} \left(1 - \operatorname{erf}\left(\frac{1}{\sqrt{2}\sigma} \right) \right) = \frac{1}{2} \left(1 - \operatorname{erf}\left(\sqrt{\frac{N}{2\,(p-d)}} \right) \right)$$

A.3 Computing network parameters given maximum accepted error

Assuming all training patterns are independent, we can use the above results to determine the minimum number of neurons which should be used for the network given the number of training patterns we want to store. In the case of image recognition, this can provide to be useful as it can give a guideline towards how to resize the images in order to bring them to the same size (patterns which are stored in a Hopfield network need to have equal lengths). It also enables us to compute the capacity of a network of given size, in order to minimise errors.

Given p and N, we deduced the probability of error for the network. Thus, conversely, we can compute the maximum ratio of p and N to ensure that the probability of error does not exceed a maximum accepted error probability.

$$P_{\text{error}} = \frac{1}{2} \left(1 - \operatorname{erf}\left(\sqrt{\frac{N}{2p}} \right) \right)$$

$$\Rightarrow \operatorname{erf} \sqrt{\frac{N}{2p}} = 1 - 2P_{\text{error}} \quad \Rightarrow \quad \sqrt{\frac{N}{2p}} = \operatorname{inverf}\left(1 - 2P_{\text{error}}\right)$$

$$\Rightarrow \frac{N}{p} = 2 \left(\operatorname{inverf}\left(1 - 2P_{\text{error}}\right) \right)^2$$

$$\frac{p}{N} = \frac{1}{2 \left(\operatorname{inverf}\left(1 - 2P_{\text{error}}\right) \right)^2} \quad \text{and} \quad N = 2p \left(\operatorname{inverf}\left(1 - 2P_{\text{error}}\right) \right)^2$$

where inverf is the inverse of the erf function.

A.4 Measurements using the error of a network for independent patterns

We will now give the reader an idea of how one should accommodate a fixed number of patterns depending on the error accepted by the application, by creating networks of appropriate size.

The following table describes the capacity of the network in terms of the error accepted, by fixing the size of the network.

The above results can be reproduced by using the Analysis module. For example, by using `ghci`:

```
*Analysis > computeErrorSuperAttractor 10 100
```

The first argument of the function is p (the number of training patterns), and the second one is N (the size of the network). A similar function can be used for a given network `computeErrorSuperAttractor`, that given a network computes the probability of error of the super attractor. The caller has to ensure that training patterns are independent in order for the computation to be correct.

p	error	N
100	0.1	165
100	0.01	542
100	0.05	271
100	0.001	955

Table A.1: Getting the minimum numbers of neurons required by the network by fixing the error and the number of patterns used to train the network.

N	error	p
1000	0.1	608
1000	0.01	184
1000	0.05	359
1000	0.001	104

Table A.2: Getting the capacity of a network by fixing the error and the size of a network.

A.5 Decreasing the error by using one super attractor

Above we described the derivation for obtaining the error of a super attractor, given its degree. We remind the reader that we did this under the assumption that all other training patterns are independent. We will now show how the error of a pattern decreases if it is made a super attractor, by varying the number of times it is presented to the network during training.

degree	p	N	error
1	20	100	$1.08 *10^{-2}$
2	20	100	$7.64*10^{-3}$
5	20	100	$4.91*10^{-3}$
8	20	100	$1.95 * 10^{-3}$
11	20	100	$4.29 * 10^{-4}$
15	20	100	$3.87 * 10^{-6}$
degree	**p**	**N**	**error**
1	50	100	$7.65 * 10^{-2}$
5	50	100	$6.80*10^{-2}$
10	50	100	$5.69*10^{-2}$
20	50	100	$3.39*10^{-2}$
30	50	100	$1.26 * 10^{-2}$
40	50	100	$7.82 * 10^{-4}$

Table A.3: Computing the network error when using a super attractor

Note that in the second table we are intentionally stressing the network so that we can notice how super attractors also affect the capacity of the network.

The above results can be reproduced by using the Analysis module. For example, by using `ghci`:

```
*Analysis > computeErrorSuperAttractorNumbers 40 50 100
```

The first argument of the function is the degree of the attractor, the second one is p and the third one is N. A similar function can be used for a given network `computeErrorSuperAttractor`, that given a network computes the probability of error of the super attractor. The caller of the function has to ensure that the training patterns contain a super attractor and that the other patterns are independent.

Appendix B

Neuron Update Optimisation

B.1 Initial Method

Originally, each update iteration of the Hopfield network involved scanning through all neurons and computing the h value. As this involves scanning through all the neurons in every step, it is clearly an $O(n)$ operation.

B.2 Optimisation

First we shuffle the list of neurons[1]. We then select the first neuron. If it is updatable then we are done. If not then continue with the second neuron, and so on. In the worst case, we will perform as bad as the initial method described above. In general, however, we will always beat or match the above algorithm as we avoid computing the h value for every node.

At the beginning of the convergence process, the patterns tend to undergo a lot of transformations, and have a large number of updatable neurons. Depending on the size of the basin of attraction in which the pattern will land, and on the journey the pattern has to go trough until it reaches the basin, the number of updatable neurons during the journey varies greatly.

As a pattern approaches convergence, the number of updatable neurons decreases (as for a fixed point, there are no updatable neurons) so we slightly reach the old performance. In any case our algorithm is still at least as good as the initial method.

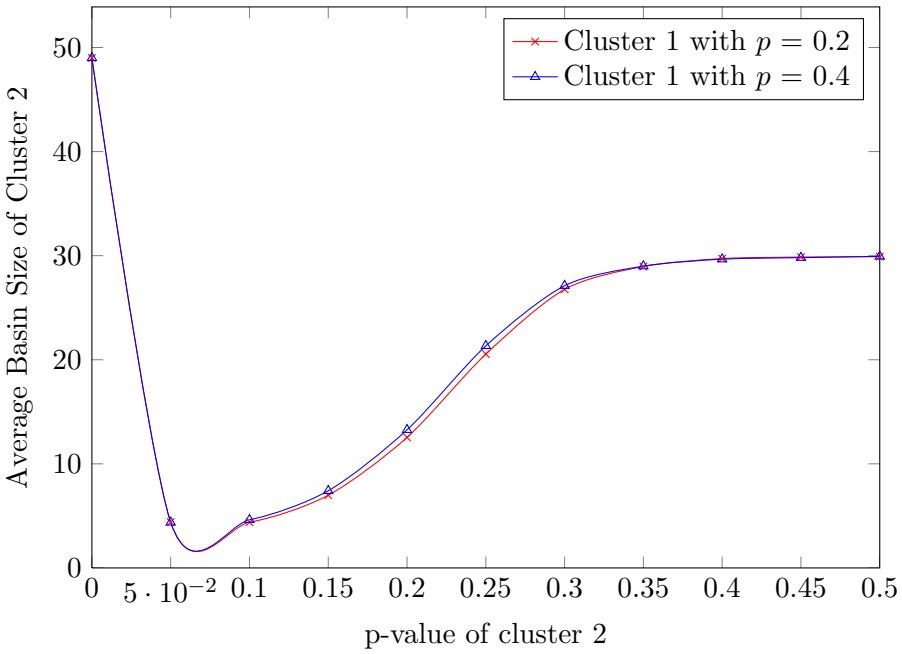
[1] Although this is an extra $O(n)$ operation, it has a negligible footprint to computing the h value

Appendix C

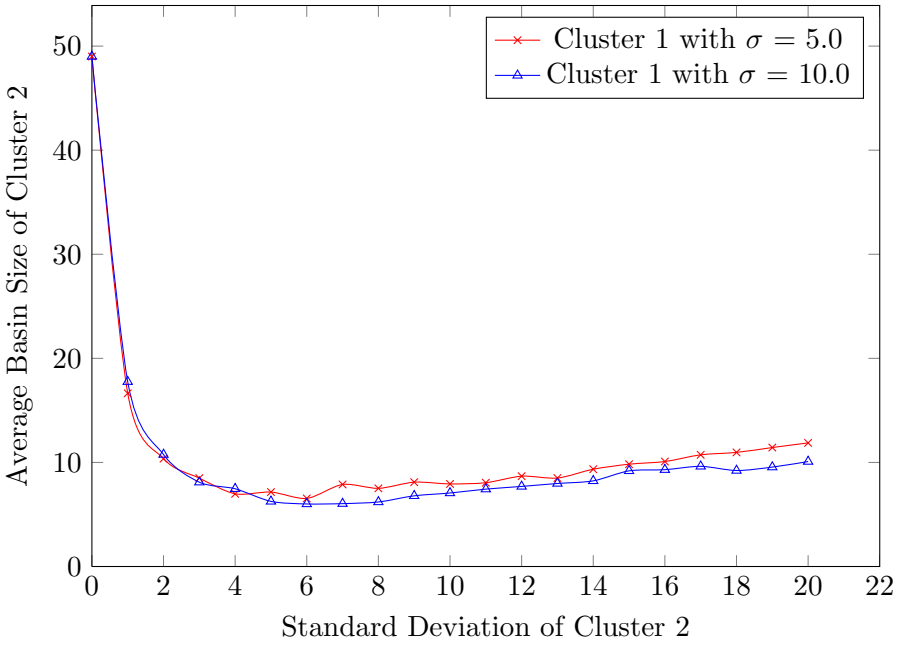
Experiments with Storkey Learning

C.1 Cluster Experiments

C.1.1 T1 Experiment



C.1.2 T2 Experiment



Appendix D

Recognition Software - Screenshots

D.1 Suspect Database

D.2 Suspect Matching

D.3 Understanding the Hopfield Model