An Analysis of the Local Optima Storage Capacity of Hopfield Network Based Fitness Function Models

Kevin Swingler(✉) and Leslie Smith

Computing Science and Maths, University of Stirling, Stirling FK9 4LA, Scotland
kms@cs.stir.ac.uk

Abstract. A Hopfield Neural Network (HNN) with a new weight update rule can be treated as a second order Estimation of Distribution Algorithm (EDA) or Fitness Function Model (FFM) for solving optimisation problems. The HNN models promising solutions and has a capacity for storing a certain number of local optima as low energy attractors. Solutions are generated by sampling the patterns stored in the attractors. The number of attractors a network can store (its capacity) has an impact on solution diversity and, consequently solution quality. This paper introduces two new HNN learning rules and presents the Hopfield EDA (HEDA), which learns weight values from samples of the fitness function. It investigates the attractor storage capacity of the HEDA and shows it to be equal to that known in the literature for a standard HNN. The relationship between HEDA capacity and linkage order is also investigated.

1 Introduction

A certain class of optimisation problem may be solved (or an attempt at solving may be made) using metaheuristics. Such problems generally have the following qualities: the search is for an optimal (or near optimal) pattern of values over a number (often many) of random variables; any candidate solution, which is an instantiation of each of those variables, has an associated score, which is its quality as a solution; a fitness function exists that takes a candidate solution and produces a score. The function (or algorithm) for calculating the score may be evaluated for any input vector, but may not be inverted to produce an input vector that would maximise the score. For this reason, the process of optimisation may be viewed as a directed search. Metaheuristics are methods for making use of the score returned by the fitness function to speed the search for good solutions when an exhaustive search would not be practical.

Most metaheuristic algorithms maintain a memory of some kind that reflects the input patterns previously chosen and the scores they received. In general such algorithms proceed by generating one or more new candidate solutions based on their memory of previous trials. Each new solution is then given a score using the fitness function and the memory is updated. The new memory is then used to

© Springer-Verlag Berlin Heidelberg 2014
N.T. Nguyen (Ed.): TCCI XVII 2014, LNCS 8790, pp. 248–271, 2014.
DOI: 10.1007/978-3-662-44994-3_13

produce new (hopefully improved) solutions and the process continues until some stopping criterion is met. One simple way to divide metaheuristic algorithms is between those that develop an explicit model and those that maintain a population of 'good' solutions in place of a model. Perhaps the best known population based method is the Genetic Algorithm (GA) [1]. A GA maintains a population of good solutions and uses them to produce new solutions that are combinations of two (sometimes more) existing solutions. Combination is performed by picking values for each variable from one or other parent to instantiate the variables in a new child solution. Better solutions have a higher probability of producing offspring than poorer ones and some offspring may be altered slightly (a process called mutation) to extend the search space. In a GA the population and the recombination process are the memory and the generative process respectively.

An alternative to maintaining a population of solutions is to build a model of some aspect of the fitness function and use that to guide the search. One well studied method is to model the probability of each variable taking each of its possible values in a good solution. As the model evolves, new candidate solutions are drawn from the model's distribution, which in turn cause the model to be updated. This approach is known as an Estimation of Distribution Algorithm (EDA) and is described in more detail in Sect. 2.

A problem that many metaheuristic algorithms face (GAs and EDAs included) is that of local optima. Local optima are globally sub-optimal points that are optimal in a local neighbourhood. That is, points from which a small change in the variables' values will not lead to an improvement in score. In terms of metaheuristic algorithms local optima are points from which the chosen algorithm will not move, even when there are better scoring solutions in other parts of the search space. For example, a population in a GA may reach a state where neither recombination nor any small mutation will produce a sufficient improvement to generate a candidate solution that is better than any in the existing population. When sampling from an EDA, a local optimum is a solution that has a high probability of being sampled, but which is not the best solution. An EDA can contain two types of local optima. Firstly, if none of the patterns that it produces are the true global optimum, then they are all local optima. Secondly, an EDA can contain many local optima — all of the high probability patterns — and one of those will score more than the others, making those others local optima compared to the model's own global optimum. In a perfect model, the EDA's global optimum will be the global optimum of the problem being tackled. In less than perfect models, it may well be a local optimum itself.

The number of local optima a fitness function contains is one measure of the difficulty of finding the global optimum. Another related measure is the size of the field of attraction around each optima. The field of attraction for an optimum is the sub-space of points from which an algorithm will make a series of steps leading to that optimum. For a simple hill climbing algorithm, the field of attraction for a local optimum, $\mathbf{x}$ is the set of points from which a hill climb will lead to $\mathbf{x}$. So called *deceptive* functions are those in which the fields of attraction of local optima are large and that for the global optimum is small [2].

A fundamental aspect of GA research considers the role played by subsets of the variables being optimised. These subsets are often called building blocks [1] or analysed as schemata [3]. For example, in a vector $\mathbf{x} = x_1 \ldots x_n$, it may be that variables x_2, x_3 and x_4 all interact in a non-linear fashion and that the effect of changing the value of any one of them can only be understood in terms of the values of the other two. This interdependency between variable subsets and the fitness function output is known as the linkage problem [4] or, sometimes in the GA literature, epistasis [5]. The number of variables in a building block is known as its order and we can talk about a problem being of order m if the order of its highest meaningful building block is m. This introduces the question of how to discover the linkage order of a function.

One way in which researchers have addressed the question of linkage order is with the use of Walsh functions [6,7]. These are described in detail in Sect. 3 and summarised here. The Walsh functions form a basis for functions over $f : \{0,1\}^n \rightarrow \mathbb{R}$. Any such function can be decomposed into a weighted sum of Walsh functions. The Walsh functions that contribute to the weighted sum are of differing orders, defined by the number of bits they contain that are set to 1. The process of performing a Walsh decomposition of a function produces the weights (known as the Walsh coefficients) associated with each Walsh function and many have values of zero. Consequently, the order of the functions with non-zero coefficients tells us about the order of interactions in the function.

These two measures of problem difficulty — fields of attraction to local optima and linkage order — are related. This paper addresses the relationship between local optima attractors and linkage order for a particular type of EDA implemented using a Hopfield Neural Network (HNN). The HNN is described in detail in Sect. 4. It can be understood variously as a neural network using an adjusted Hebbian learning rule and McCulloch-Pitts neurons, or as a Markov random field or as a second order EDA. It is well known [8] that HNNs have a certain capacity for storing patterns in memory and this capacity holds for the number of local optima a network can learn. The important point to note is that in the standard HNN, only second order linkages are learned. It is not the case, however, that the HNN can only find all the attractors in problems of order 2 or below. We show that a HNN can discover the local optima of functions with higher order linkages, up to some capacity, and investigate the relationship between linkage order, network capacity and local optima count.

The main contributions of this paper are the adaptation of a Hopfield network to learn functions with real valued outputs and an analysis of the number of local optima such networks can represent at one time. An algorithm for improving the capacity of a Hopfield network is also adapted and its capacity analysed. Several functions are analysed in terms of their linkage order and the number of local attractors they contain. These results are then related to the capacity of Hopfield networks and their ability to capture the local optima in higher order functions. The paper suggests a parallel between the number of attractor states in the network and population diversity in a GA. This should be of interest to researchers developing multi-variate EDAs as it highlights the need to manage the attractors of an EDA and demonstrates how all attractors in a model degrade

once capacity is exceeded. The main focus of the paper is on the second of these contributions—the analysis of attractor capacity. An analysis of the ability of the method to improve optimisation requires the study of algorithms for evolving models and solutions and a large number of experiments, which are all the focus of a future paper.

The paper is organised as follows. Sections 2, 3 and 4 introduce EDAs, Walsh functions and Hopfield networks respectively. Sections 5 and 6 describe a Hopfield EDA (HEDA) and presents two learning rules: one based on a standard Hebbian update and one designed to improve network capacity. Section 7 describes a set of experiments and an analysis of network size, capacity and the time taken during learning. Section 8 analyses the weights of a HEDA and Sect. 9 offers some conclusions and discusses future work.

1.1 Scope

The functions discussed in this paper are real valued functions of binary vectors, that is:

$$f : \{-1, 1\}^n \rightarrow \mathbb{R} \tag{1}$$

A single candidate solution is a point in n-dimensional binary space defined by a vector $\mathbf{c}$ of elements c_i that can take binary values:

$$\mathbf{c} = c_1, \ldots, c_n \qquad c_i \in \{-1, 1\} \tag{2}$$

The function $f(\mathbf{c})$ that guides the search is known as the fitness function.

2 Estimation of Distribution Algorithms

A common class of model based optimisation methods are the Estimation of Distribution Algorithms (EDAs). Rather than maintain individual searches, an EDA attempts to model the probability of a value or sub-pattern appearing in a high scoring candidate solution. Reasons for building a fitness model include the advantages gained from being able to analyse the model to better understand the problem to be optimised [9] and the improved speed of estimating fitness function output rather than making a time consuming calculation for every required evaluation.

The simplest EDA models the marginal probability of each variable taking each of its possible values in the optimal solution. Population Based Incremental Learning (PBIL) [10] is an example of such a method. PBIL works by sampling from the set of possible solutions and maintaining a marginal probability distribution for each variable. PBIL is a population based search as the probabilities are updated based on the best scoring members of each generation of the population. Subsequent populations are generated based on the probabilities in the distribution. PBIL makes use of a learning rate to smooth the evolution of the probabilities. For solving problems with binary valued variables, PBIL's update rule is:

$$P(p_i = 1) \leftarrow (1 - \alpha)P(p_i = 1) + \alpha\rho(p_i = 1) \tag{3}$$

where $\leftarrow$ indicates assignment, $P(p_i = 1)$ is the probability that $p_i = 1$ in the model, and $\rho(p_i = 1)$ is the probability that $p_i = 1$ amongst the best of the current population. α is the learning rate.

PBIL is a first order method as it models each variable independently. It will solve first order problems as they have no local optima to trap it. For higher order problems, it may find the global optimum or it may become trapped in a local optimum. The evolutionary aspect of PBIL ensures that it converges on an optimum of some kind (local or global). Other univariate algorithms include the compact Genetic Algorithm (cGA) [11] and the Univariate Marginal Distribution Algorithm (UMDA) [12].

Early attempts at capturing second order interactions included the Mutual Information Maximising Input Clustering algorithm (MIMIC) [13]. This algorithm imposes an ordering on the variables and links them in a chain so that each variable (except the last in the chain) is linked to exactly one other. The ordering is discovered using a greedy algorithm (a full search for the correct ordering is NP-complete), and so is not guaranteed to find the optimal chain. Another pairwise method is the Bivariate Marginal Distribution Algorithm (BMDA) [14], which models second order linkage in a forest (a set of independent dependency trees). Higher order interactions are learned by the Bayesian Optimisation Algorithm (BOA) [15], which builds a Bayesian network to attempt to capture the joint distribution of values in a population of promising solutions. Again, building a correct Bayesian network is an NP-complete problem, so heuristics are needed to build the network in sensible time. More recently, the Distribution Estimation Using Markov networks (DEUM) algorithm [16] has modelled second and higher order interactions using Markov random fields. DEUM models conditional probabilities and generates new samples using Gibbs sampling.

Most EDA approaches consist of a method for representing the distribution and a method for evolving a solution. The evolution of a solution generally follows a pattern of generating a population of candidate solutions from the current model, updating the model based on the quality of those candidates by either learning only the most fit or by learning them all with a weight that depends on the fitness score. The process is then repeated until the algorithm finds a good solution or converges. There are a number of choices to be made when designing the evolutionary algorithm: whether to learn a subset of the population or use the score of every member; the size of each generation, which effects the accuracy of the model; whether to start a new model at each generation or to add to the existing one; and what schedule, learning rate or forgetting factor to use if a model persists from generation to generation. These factors greatly affect the quality of an EDA optimisation algorithm, but they are not the subject of this paper, which demonstrates how a second order EDA can be built from a HNN and then makes use of the fact that much is understood of the capacity and function of such networks.

These methods all share a common feature: they do not attempt to model the distribution of values in all good solutions, just a small number of them. In the

case of the univariate methods, it should be clear that the number of solutions the distribution can model is just one: the pattern produced by picking the value with the highest probability from each variable. In multivariate models, the number of different promising patterns that can be stored at any one time is greater than one. This paper investigates this capacity for bivariate models.

3 Walsh Functions

Walsh functions [17] form a basis for functions of binary vectors. Any function $f : \{0,1\}^n \rightarrow \mathbb{R}$ can be represented as a weighted sum of Walsh functions. The contribution of each Walsh function to the sum is determined by its Walsh coefficient.

3.1 Generating the Walsh Functions

To decompose a function of n variables, Walsh functions of length 2^n are used, denoted ψ_j where $j = 0 \ldots 2^n - 1$. There are 2^n such functions, each represented by a string of 2^n bits: $\psi_j(c)$ where $c = 0 \ldots 2^n - 1$. $\psi_j(c)$ is the cth bit in the jth Walsh function. They are calculated in a bit wise fashion from the binary representation of their indices, c and j. Note the slight abuse of notation: j is an integer index, $\mathbf{j}$ is the binary representation of j and j_i is the ith bit of $\mathbf{j}$, counting from the right. For example, $j = 3$, $\mathbf{j} = 011$, $j_3 = 0$. The same applies to c. To calculate $\psi_j(c)$, first re-code the binary representation of c so that 1 becomes -1 and 0 becomes 1. This slightly counter-intuitive re-coding allows multiplication to perform the XOR function. This can be done bitwise using:

$$y_i \leftarrow -(2c_i - 1) \tag{4}$$

Then use the recoded vector $\mathbf{y}$ and the binary representation of $\mathbf{j}$ to calculate each bit $\psi_j(y)$ as

$$\psi_j(y) = \prod_{i=1}^{n} y_i^{j_i} \tag{5}$$

where $j_i \in \{0, 1\}$ so $y_i^{j_i} = y_i$ when $j_i = 1$ and $y_i^{j_i} = 1$ when $j_i = 0$. The binary word $\mathbf{j}$ acts as a mask to determine which values $y_i \in \{-1, 1\}$ are included in the product. Figure 1 shows a pictorial representation of the order 3 Walsh functions.

3.2 Calculating the Walsh Coefficients

The Walsh transform of an n-bit function produces 2^n Walsh coefficients, ω_j where $j = 0 \ldots 2^n - 1$. Each coefficient is calculated as follows:

$$\omega_j = \frac{1}{2^n} \sum_{c=0}^{2^n-1} f(\mathbf{c})\psi_j(c) \tag{6}$$

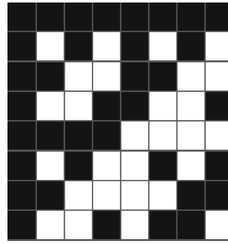

Fig. 1. A pictorial representation of the third order Walsh functions with black squares representing 1 and white squares −1.

where $f(\mathbf{c})$ is the real valued output of the fitness function when the input variables are instantiated with the values from the binary representation of c. Note that for $f : \{-1, 1\}^n$, each 0 in the binary word must first be converted to −1.

3.3 Stating the Fitness Function as a Walsh Function

The function $f(\mathbf{c})$ can now be restated as a Walsh sum:

$$f(\mathbf{c}) = \sum_{j=0}^{2^n} \omega_j \psi_j(c) \tag{7}$$

where c is the integer represented by the binary word $\mathbf{c}$, with the same allowance for converting $0 \leftarrow -1$ if required when converting from $\mathbf{c}$ to the bit index c.

3.4 Analysis of Walsh Coefficients

We can now directly introduce the concept of linkage order with respect to Walsh functions. A Walsh decomposition of an n-bit function leads to 2^n Walsh coefficients, $\omega_i (i = 0 \ldots 2^n - 1)$. The index i determines how the coefficient ω_i is calculated (see Eq. 6). It also determines the linkage order of the coefficient. Let the binary equivalent of the index i of ω_i be $\mathbf{i}$, which acts as a mask, selecting bit positions where there is a 1 in $\mathbf{i}$. Counting the number of 1s in $\mathbf{i}$ tells you the linkage order of ω_i. For example, ω_0 is of zero order and ω_3 (011) is second order. Consider the full results of a Walsh analysis of a first order binary problem over three bits where the target pattern is 101 and the fitness function is an inverse Hamming distance such that $f(101) = 1$ and $f(010) = 0$. The first order coefficients, ω_1, ω_2 and ω_4 all have none-zero values and the higher order coefficients are all zero.

Table 1 shows how to extract the first order optimum from the Walsh coefficients directly. The location of the 1 bit in the first order index, $\mathbf{i}$ of ω_i corresponds to the location in the optimum whose value is determined by ω_i. If ω_i is positive, the optimum contains a zero at that location and if it is negative,

Table 1. Walsh Coefficients and linkage order for $f(\mathbf{c}) = 1 - Hamming(c, 101)$

Coefficient	Index	Order
$\omega_0 = 0.5$	000	0
$\omega_1 = -0.167$	001	1
$\omega_2 = 0.167$	010	1
$\omega_4 = -0.167$	100	1
$\omega_3 = 0$	011	2
$\omega_5 = 0$	101	2
$\omega_6 = 0$	110	2
$\omega_7 = 0$	111	3

the optimum contains a 1. Looking at Table 1, we see that ω_1 is negative, ω_2 is positive and ω_4 is negative, so the optimum must be at 101.

A full Walsh decomposition requires 2^n function evaluations and so is not a practical method for solving large optimisation problems. It is, however, a useful tool for understanding concepts of linkage on small, toy problems.

4 Hopfield Networks

Hopfield networks [18] are able to store patterns as point attractors in n dimensional binary space and recall them in response to partial or degraded versions of stored patterns. For this reason, they are known as content addressable memories where each memory is a point attractor for nearby, similar patterns. Traditionally, known patterns are loaded directly into the network (see the learning rule 10 below), but in this paper we investigate the use of a Hopfield network to discover point attractors by sampling from a fitness function. A Hopfield network is a neural network consisting of n simple connected processing units. The values the units take are represented by a vector, $\mathbf{u}$:

$$\mathbf{u} = u_0, \ldots, u_{n-1} \qquad u_i \in \{-1, 1\} \tag{8}$$

The processing units are connected by weighted connections:

$$W = [w_{ij}] \tag{9}$$

where w_{ij} is the strength of the connection from unit i to unit j. Units are not connected to themselves, i.e. $w_{ii} = 0$ and connections are symmetrical, i.e. $w_{ij} = w_{ji}$. The values of the weighted connections define the point attractors and learning in a standard Hopfield network takes place by setting the pattern to be learned using formula 11 and applying the Hebbian weight update rule:

$$w_{ij} \leftarrow w_{ij} + u_i u_j \quad \forall i \neq j \tag{10}$$

A single pattern, **c** is set by

$$\forall i \quad u_i \leftarrow c_i \tag{11}$$

Pattern recall is performed by allowing the network to settle to an attractor state determined by the values of its weights. The unit update rule during settling is

$$a_i \leftarrow \sum_{j=0}^{n-1} w_{ji} u_j \tag{12}$$

where a_i is a temporary activation value, following which the unit's value is capped by a threshold, θ, such that:

$$u_i \leftarrow \begin{cases} 1 & \text{if } a_i > \theta \\ -1 & \text{otherwise} \end{cases} \tag{13}$$

In this paper, we will always use $\theta = 0$. The process of settling repeatedly uses the unit update rule of formulae 12 and 13 for a randomly selected unit in the network until no update produces a change in unit values. At that point, the network is said to have settled. The symmetrical weights and zero self-connections mean that the network is a Lyapunov function, which guarantees that the network will settle to one of its fixed points from any starting point. With the above restrictions in place, the network has an energy function that determines the set of possible stable states into which it will settle. The energy function is defined as:

$$E = -\frac{1}{2} \sum_{i,j} w_{ij} u_i u_j \tag{14}$$

Settling the network, by formulae 12 and 13 produces a pattern corresponding to a local minimum of E in Eq. 14. Hopfield networks have been used to solve optimization tasks such as the travelling salesman problem [19] but weights are set by an analysis of the problem rather than by learning. Other examples of hand built Hopfield optimisers include [20], in which the authors comment on the lack of a method for finding the right set of weights for an arbitrary optimisation problem. In the next section, we show how random patterns and a fitness function can be used to train a Hopfield network as a search technique.

5 Hopfield EDAs

We define a Hopfield EDA (HEDA) as an EDA implemented by means of a Hopfield neural network. This section describes the training and use of a HEDA. Figure 2 shows a four neuron HEDA with the units labelled u_i and weights in one direction labelled $W_{i,j}$.

5.1 Training a HEDA

In this section we describe a method for training a HEDA. The principles apply equally to HEDAs of higher order. During learning, candidate solutions are generated randomly one at a time. Each candidate solution is evaluated using the

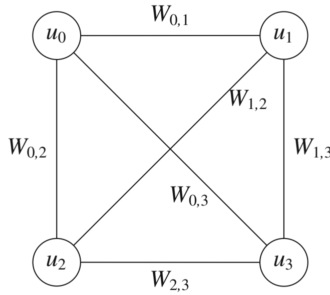

Fig. 2. A four neuron HEDA with units u_i and weights $W_{i,j}$. Due to weight symmetry, only half the weights are shown as each $W_{i,j} = W_{j,i}$.

fitness function and the result is used as a learning rate in the Hebbian weight update rule (see update rule 15). Consequently, each pattern is learned with a different strength, which reflects its quality as a solution.

5.2 The New Weight Update Rule

Hopfield networks have a limited capacity for storing patterns. If a number of patterns greater than this capacity are learned, patterns interfere with each other producing spurious states, which are a combination of more than one pattern. To learn the point attractors of local optima without ever sampling those points, we need to create spurious states that are a combination of lower points. We do this by over-filling a Hopfield network with samples and introducing a strength of learning so that higher scoring patterns contribute more to the new spurious states. This yields a simple modification to the Hebbian rule:

$$w_{ij} \leftarrow w_{ij} + f(\mathbf{c})u_i u_j \quad i \neq j \tag{15}$$

where $\mathbf{c}$ is the candidate solution to be learned and $f(\mathbf{c})$ is the output of the fitness function given $\mathbf{c}$. This has the effect of learning high scoring second order sub-patterns more than lower scoring ones. Note that due to the symmetry of the weight connections, each attractor has an associated inverse pattern that is also an attractor. The means that both the pattern and its inverse may need to be scored to tell the solutions apart from their inverse twins.

5.3 The Learning Algorithm

The simplest version of the learning algorithm simply builds the network from samples of $\mathbf{c}$ and $f(\mathbf{c})$ and proceeds as follows:

1. Set up a Hopfield network with W_{ij}=0 for all i, j
2. Repeat the following until one or more stopping criteria are met
   (a) Generate a random pattern, $\mathbf{c}$, where each c_i has an equal probability of being set to 1 or -1

(b) Calculate $f(\mathbf{c})$
(c) Load $\mathbf{c}$ into the network's neurons using formula 11
(d) Update the weight matrix W using the learning rule in formula 15
(e) Sample attractor states (local optima) from the network and keep the best found
(f) Stop when a pattern of required quality has been found or when the attractor states become stable or the network reaches capacity.

There are a great many improvements that can be made to this simple algorithm. The sampling can be done in a number of different ways, for example. Rather than sample after each single weight update, sampling could occur less frequently. Also, the sampled 'good' solutions from local optima could be used to drive the choice of new candidate solutions in a number of ways (as it does in an EDA). A degree of forgetting could also be introduced, either by starting a new HEDA with zeroed weights for each new set of candidate solutions, or by something less drastic such as dividing the weights by a constant. These questions are outside the scope of this paper, but may be informed by an understanding of the capacity of the HEDA to store a number of local attractors. In particular step 2e will be limited by the number of attractors the HEDA contains.

5.4 Sampling a HEDA Model

During the search process, new candidate solutions are generated by sampling the HEDA. The sampling process may be carried out in a number of ways. This paper is more concerned with network capacity than with the details of sampling methods, but some concepts are outlined here. Local optima can be sampled by picking a random pattern or a pattern from the current population if a population based search is being used and loading it into the network using Eq. 11. The network is then settled to a local optima by repeatedly applying the update rule 12 to neurons picked in random order until no neuron produces a change in its output value over an exhaustive sweep of the network. Neurons can be treated stochastically by replacing the activation function with a probability based calculation. This turns the HNN into a Boltzmann machine [21] and allows simulated annealing to be used as the search technique.

6 A Learning Rule for Improving Capacity

Storkey [22] introduced a new learning rule for Hopfield networks that increased the capacity of a network compared to using the Hebbian rule. The new weight update rule is:

$$w_{ij} \leftarrow w_{ij} + \frac{1}{n}u_i u_j - \frac{1}{n}u_i h_{ji} - \frac{1}{n}u_j h_{ij} \tag{16}$$

where

$$h_{ij} \leftarrow \sum_{k \neq i,j} w_{ik} u_k \tag{17}$$

The new terms, h_{ji} and h_{ij} have the effect of creating a local field around w_{ij} that reduces the lower order noise brought about by the interaction of different attractors.

To use this learning rule in a HEDA, we make the following alterations to the update rules:

$$w_{ij} \leftarrow w_{ij} + \frac{1}{n}(u_i u_j - u_i h_{ji} - u_j h_{ij}) f(p) \tag{18}$$

and

$$h_{ij} \leftarrow \sum_{k \neq i,j} w_{ik} u_k \beta \tag{19}$$

where $\beta < 1$ is a discount parameter that controls how much damping is applied to the learning rule and which keeps the weights at reasonable values.

7 Experimental Results

This paper investigates the effects of the number of local optima in a fitness function and the capacity of a HEDA to represent them, rather than the effectiveness of the HEDA as an optimisation tool in its own right. The following experiments reflect this focus.

7.1 Experimental Functions

We build functions with a fixed number of local attractors using a method based on nearest Hamming distance. In this method, a number of target patterns are chosen as the local optima and the function is evaluated by calculating the Hamming distance to the closest of the set of target patterns. Consequently, the number of local optima equals the number of target patterns. The set of target patterns are denoted as the set **T**:

$$\mathbf{T} = \{t_1, \ldots, t_s\} \tag{20}$$

We then define the fitness function as one minus the normalised Hamming distance between **c** and each target pattern t_j in **T**:

$$f(\mathbf{c}, t_j) = 1 - \sum_{i=1}^{n} \frac{\delta_{c_i, t_{ji}}}{n} \tag{21}$$

where t_{ji} is element i of target j and $\delta_{c_j, t_{ji}}$ is the Kronecker delta function between pattern element i in t_j and its equivalent in **c**. We take the score of a single pattern to be the maximal score of all the members of the target set.

$$f(\mathbf{c}, \mathbf{T}) = max_{j=1 \ldots s}(f(\mathbf{c}, t_j)) \tag{22}$$

This method of building the fitness function relates to existing research on HNN capacity, which is often based on the capacity for storing random patterns.

7.2 HEDA Capacity Experiments

This section investigates the capacity of the HEDA, which is the number of distinct attractors it can model. As each attractor is a single local optimum, the capacity of the network determines the number of local optima a HEDA can represent at any one time. Depending on how the HEDA is being used, this has a number of consequences. Most EDA approaches to optimisation try to model the distribution of 'promising solutions'. A local optima in the search space represents a single neighbourhood of promising solutions. Consequently, the ability to model a number of different local optima requires the EDA to hold a number of local attractors. A univariate EDA has a single attractor and multivariate EDAs have higher capacities.

The literature on the capacity of a Hopfield network has generally concentrated on the capacity for storing random patterns. A pattern is deemed to be successfully stored in a Hopfield network if the pattern of activity corresponding to that pattern is an attractor point in the network. This is tested by setting the chosen pattern as a starting point using Eq. 11 and then settling the network using Eq. 12. If the network does not move away from the attractor point, then the pattern is still in its memory. Other nearby points will also cause the network to settle to the same attractor point, depending on the size of its basin of attraction. In terms of storing random patterns, [8] states that the capacity of a HNN is $n/(2 \ln n)$ where n is the number of neurons in the network. This is the figure we will be using for our analysis in this paper.

This set of experiments compares the capacity of a normally trained Hopfield network with the search capacity of a HEDA. We will compare two learning rules (Hebbian and Storkey). The experiments are repeated many times, all using randomly generated target patterns where each element has an equal probability of being $+1$ or -1.

7.3 Experiment 1: Hebbian HEDA Capacity

Experiment 1 compares Hebbian trained Hopfield networks with their equivalent HEDA models. The aim is to discover whether or not the HEDA model can achieve the capacity of the Hopfield network. Hopfield networks were trained on patterns using standard Hebbian learning, with one pattern at a time being added until the network's capacity was exceeded. At this point, the learned patterns were set to be the targets for the HEDA search using Eqs. 21 and 22 and the network's weights were reset.

100 repeated trials were made training HEDA networks ranging in size from 10 to 100 units in steps of 5. For each trial, the capacity of the trained network, the number of those patterns discovered by global searching and the time taken to find them all (or give up short) were recorded.

Results. Regardless of the capacity or size of the Hopfield network, the HEDA search was always able to discover every pattern learned during the capacity filling stage of the test. From this, we conclude that the capacity of a HEDA for

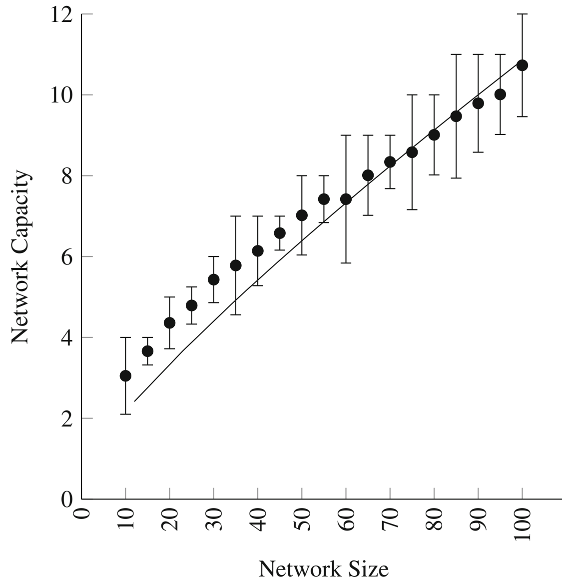

Fig. 3. The mean and inter-quartile range of the capacity of HEDA networks of varying sizes and the theoretic capacity of similar HNNs (single line).

storing local optima when searching for a set of random targets is the same as the capacity of the equivalent Hopfield network.

Figure 3 shows the relationship between HEDA network size and capacity. The spread of capacity values is wide, varying with the level of interdependence between the random patterns. The chart shows the mean and the inter-quartile range of capacity for each network size. The solid line shows the theoretical capacity of HNNs by size.

In these simple examples, no evolution takes place; the network is not used to generate new populations. This paper is not concerned with algorithms to improve search performance, it is concerned with memory capacity in EDAs (specifically the HEDA). However, it is instructive to investigate the relationship between the number of local optima a HEDA can store and the number of uniform random samples required to model them. To that end, the number of samples that were required to allow the HEDA to identify all of the local optima in the experiments above was recorded. Figure 4 shows the number of samples required to find all the local optima of a function plotted against network capacity. By curve fitting the data shown, we find that the number of samples required to find all local optima is quadratic with number of such optima.

In fact, as the search space grows, the number of iterations required to model every local optima, as a proportion of the size of search space diminishes exponentially. For networks of size 100, the search space has 2^{100} possible states and the HEDA is able to find all of the targets in an average of around 355,000 samples. That is a sample set consisting of 2.8×10^{-25} of all possible patterns.

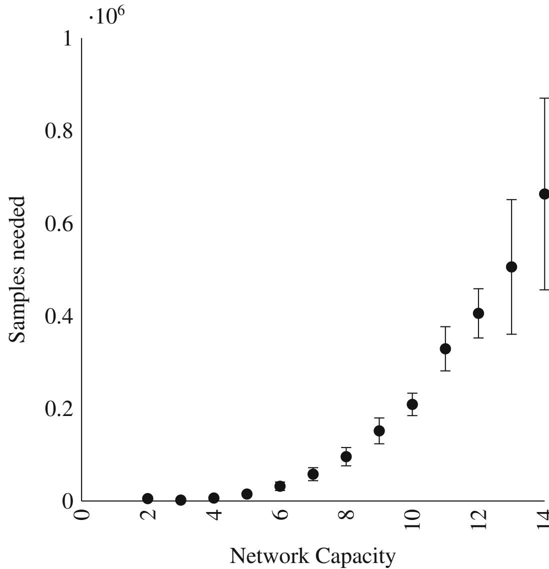

Fig. 4. The mean and inter-quartile range of the number of samples needed to find all local optima in a HEDA filled to capacity, plotted against the number of patterns to find.

Experiment 2: Storkey HEDA Capacity. As mentioned above, [22] suggest an alternative to the Hebbian learning rule that increases the capacity of a Hopfield network. This new learning rule can be used to increase the number of attractors in a HEDA and so increase the number of local optima it is able to model. A Hopfield network trained with Storkey's learning rule has a capacity of $n/\sqrt{2 \ln n}$. In experiment 2 we repeat experiment 1 but use the Storkey learning rule rather than the Hebbian version. The experimental procedure is the same.

Results. As with the Hebbian learning, the Storkey trained HEDA search was always able to discover every pattern learned during the capacity filling stage of the test. This shows that the improved learning rule will deliver the increased capacity for capturing local optima that we sought. The cost of this capacity is a far slower learning algorithm, however as Eqs. 18 and 19 have more terms to evaluate. Figure 5 shows the relationship between Hopfield network size and capacity for the Storkey trained network.

Figure 6 shows the number of samples required to find all the local optima plotted against network size when using the Storkey rule. Again, we see that search iterations increase quadratically with network capacity.

7.4 Linkage Order and Network Capacity

Section 7.2 described how a HNN and a HEDA have a capacity for storing a number of attractors, or local optima. In this section, we investigate the relationship between network size, network capacity (in terms of attractor states),

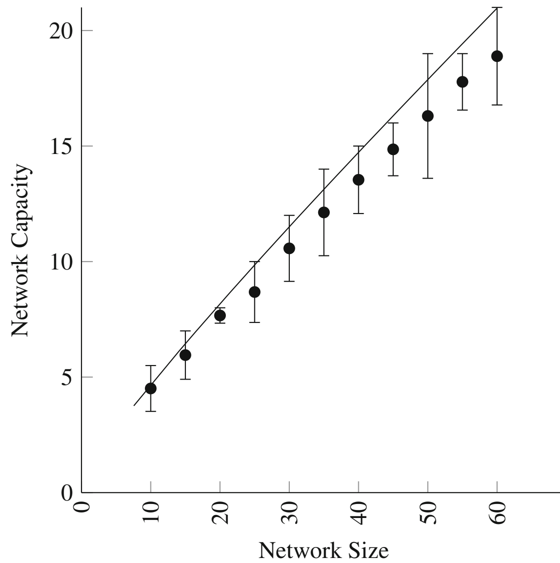

Fig. 5. The mean and inter-quartile range of the capacity of Storkey trained HEDA models and their theoretical limit (single line).

and the highest order of linkage interactions in the fitness function. In these experiments, a standard Hopfield network is trained incrementally on patterns until the addition of a new pattern causes one of the previous patterns to be forgotten, that is, the pattern is no longer an attractor state. This is tested by setting each target pattern as an input to the network, and then settling the network to a local attractor. If the network moves away from its starting point, then that point is no longer an attractor.

Once the network has reached capacity, the set of patterns that it has learned are used as the local optima in a Hamming function as in Eq. 22. The Walsh decomposition of this function is calculated and the highest order weight is recorded. This process generates pairs of numbers: the network capacity and the highest order of the function whose local optima are the patterns that fill that capacity.

Figure 7 shows the results of these experiments as a set of histograms, one for each network capacity from 2 to 5. A Hopfield network with capacity m has learned all the attractors in a function with m local optima using the standard Hebbian rule. This function undergoes a Walsh decomposition and the resulting coefficients have a maximal order at which the coefficient is non-zero. This highest order is recorded and counted for representation in the histograms. None of the networks of order m had a highest order below m and the larger the capacity, the more often the function had a larger highest order of interaction. It is clear from the histograms that many high order functions may have their attractors represented by a second order Hopfield network.

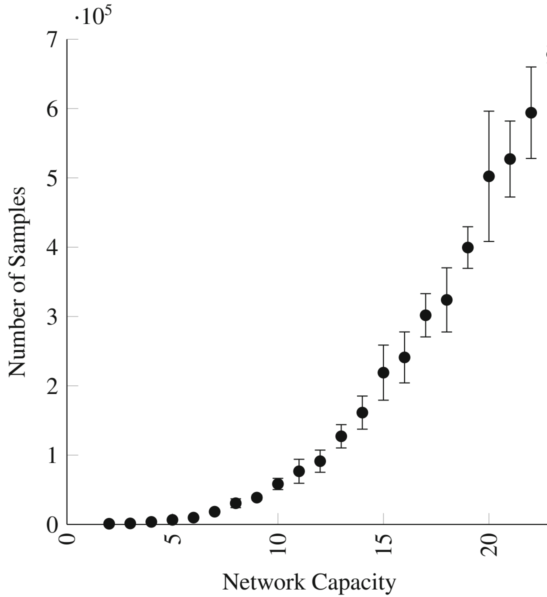

Fig. 6. The mean and inter-quartile range of the number of samples needed to find all local optima in a Hopfield network filled to capacity using the Storkey learning rule, plotted against the number of patterns to find.

8 Analysis of Network Weights

This section describes an analysis of the weights of a trained HEDA. Section 3 introduced Walsh functions and in Sect. 7.3 they were used in the analysis of fitness function linkage order. This section describes the equivalence between the weights of a HEDA and the second order Walsh coefficients. The important finding is that the weights of an exhaustively trained HEDA, $\mathbf{W}$ are equal to the second order Walsh coefficients, as stated in Eq. 23.

$$W_{ij} = \omega_c \tag{23}$$

where c is the integer obtained by constructing a binary word of n bits, setting every bit to zero except the two at indices i and j, and converting the resulting word to an integer with the standard place encoding method. The binary word, $\mathbf{b}(i,j)$ is constructed one bit at a time where $b_k(i,j)$ is the kth bit (least significant first) of the word constructed for weight W_{ij}.

$$b_k(i,j) = \begin{cases} 1, & \text{if } k = i \text{ or } k = j \\ 0, & \text{otherwise} \end{cases} \tag{24}$$

Then c is calculated as

$$c \leftarrow \sum_{k=0}^{n-1} 2^{b_k} \tag{25}$$

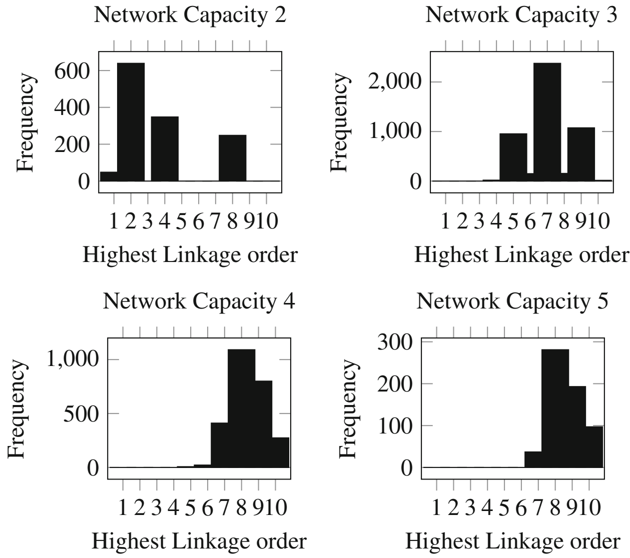

Fig. 7. Histograms showing the frequency of the highest linkage order across 10,000 trials, organised by Hopfield network capacity. Networks are trained with the standard Hebbian rule. Networks with capacity greater than 5 require a number of units greater than that for which it is possible to run multiple Walsh decompositions.

An exhaustively trained HEDA is built by generating every possible pattern across the input space, scoring each one, and training the HEDA on the resulting input/output pairs. After training, the weights are all divided by the number of patterns learned (i.e. 2^n), so the learning rule for each weight, given every sample is

$$W_{ij} \leftarrow \frac{1}{2^n} \sum_{\mathbf{c} \in \{-1,1\}^n} f(\mathbf{c}) c_i c_j \tag{26}$$

There is a clear parallel between Eqs. 26 and 6 and between Eqs. 14 and 7, which shows that the HEDA's energy function is a second order approximation of the learned function. Section 7.3 showed that second order HEDAs can capture the local optima of higher order functions, but Eq. 7 suggests that the inclusion of lower order weights in the HEDA might also be of use. First order weights are the equivalent of adding bias weights to each neuron in a Hopfield network. Let the first order weights be a vector, $\mathbf{v}$ of size n. To introduce ω_0 requires the addition of a zero order weight to the HEDA, w_0, which has the rather un-neural quality of being connected to no neurons. However, if these weights are included in the HEDA energy function, then the approximation to the fitness function becomes more accurate. Now, an approximation to $f(\mathbf{c})$ can be calculated as

$$f(\mathbf{c}) = w_0 - \sum_i v_i u_i + \sum_{i,j} w_{ij} u_i u_j \tag{27}$$

where w_{ij} are the second order weights, v_i are the first order weights and w_0 is the single order zero weight. Calculating the first and second order weights during learning is done as follows.

$$w_0 \leftarrow w_0 + f(\mathbf{c}) \tag{28}$$

and

$$v_i \leftarrow v_i + f(\mathbf{c})u_i \tag{29}$$

After dividing by the number of training examples, w_0 becomes the mean of all the function outputs, i.e. $w_0 = \langle f(\mathbf{x}) \rangle$.

8.1 Comparing Network Energy to the Fitness Function

A series of experiments compared the energy function of the HEDA, as calculated in Eq. 27 to the true output of the fitness function 22 for a variable number of target patterns. As expected, with few target patterns, the output of the energy function matched the output of the fitness function. As the number of targets (local optima) rose, the second order approximation of the HEDA became less accurate and its capacity for correctly scoring local optima above other patterns diminished. Figure 8 shows some results of these experiments. Each graph shows the energy output from a HEDA calculated using Eq. 27 plotted against the target output from Eq. 22. Each graph in Fig. 8 was produced with a different set of target patterns. It is clear from the figures that the HEDA's ability to model the function and capture the optima reduces as the number of optima increases.

The graphs showing functions with one and two local optima demonstrate that the function output calculated using Eq. 27 is equal to the true function output for every different input pattern. The graph for four local optima shows that three of those optima have been captured by the network, and that there is some difference between the network output and the function output. The last plot, with eight local optima, shows the network performance degraded past the point where it is useful. The plot shows a number of spurious optima — maximal in the network output but not in the function and also shows that the true optima have not been recorded. This illustrates a central point to this paper: if the population of attractors being modelled is too large, the ability of the HEDA (and other similar EDAs) degrades quickly to the point where all the local optima are lost. The number of quality of the attractors needs careful management.

8.2 Comparing HEDA Weights to HNN Weights

A number of sampling methods for training a HEDA have been mentioned in this paper. The simplest is uniform random sampling. A more directed evolutionary approach is to let the current attractors of the HEDA guide the sampling of the next generation of candidate solutions. The exhaustive method used above

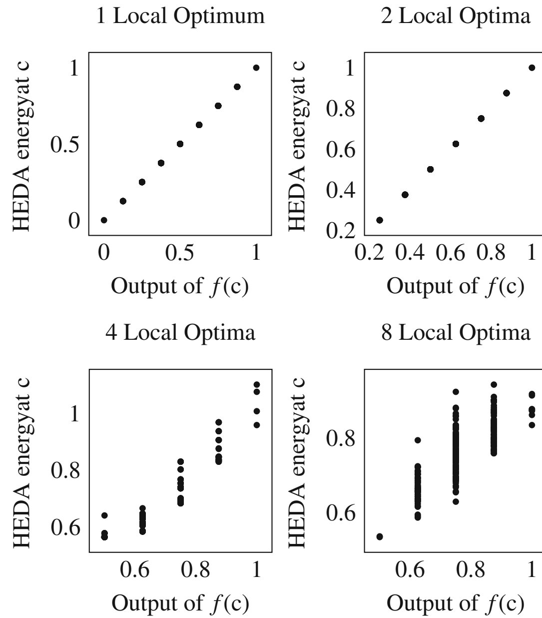

Fig. 8. HEDA energy plotted against fitness function output for functions with different numbers of local optima

trains the HEDA on every possible candidate solution, which is not of interest in terms of optimisation, but is useful in revealing structure in the weights.

A standard HNN can be viewed as an extreme example of the second method: an evolutionary approach where only the local optima of the fitness function are sampled and learned. In the simplest case, a HNN with a single pattern in its memory has learned a function where $f(\mathbf{c}) = 0$ everywhere except where $\mathbf{c}$ equals the single learned pattern, where $f(\mathbf{c}) = 1$. In this case, Eq. 10 differs only from Eq. 15 by the constant term $\frac{1}{2^n}$ as nothing is learned where $f(\mathbf{c}) = 0$ and the pattern is learned once when $f(\mathbf{c}) = 1$. In terms of the attractors, only the relative magnitude and the sign of the weights are important. Scaling all the weights in a network by dividing by a constant (in this case, 2^n) makes no difference to the location of the attractors. The function described here has interactions all the way up to order n, so although the network can model the attractor correctly, it cannot reproduce the function correctly. Adding further attractors, up to the capacity of the network, maintains the equivalence. The HEDA weights are simply a multiple of the HNN weights.

Now consider a similar function, where the Hamming measure of Eq. 21 is used with a single target pattern. When sampling $f(\mathbf{c})$ at random and updating the HEDA weights using Eq. 15 the first pattern sampled will become an attractor point instantly, regardless of its score. Subsequent samples will first add and then move the attractor points until the weights combine to produce a 'spurious' attractor that happens to be the global optimum (this will happen

without the input pattern that leads to that optimum being sampled). At this point, the weights will represent a second order approximation to the first order Hamming function - just as they would in a HNN trained on the single pattern. In this example, continuing to train would eventually move all the second order weights to zero and the solution would only remain in the first order bias weights (if they were included).

8.3 Comparison with Other EDAs

The purpose of this paper is to consider the capacity of a second order EDA for storing local optima as attractor points. It is not to demonstrate that one EDA structure or optimisation algorithm is better than any other. It is instructive, however, to compare the HEDA approach to other EDAs to demonstrate that the measurement of capacity generalises across methods. Starting with the simplest first order EDA (such as PBIL or UMDA), it is clear that such models have a capacity of one—the single most probable pattern, which maximises $\prod_{i=1}^{n} P(c_i)$. The compact GA (cGA) [11] overcomes the limitation of a univariate EDA by building a marginal probability model (MPM) that is a product of joint probabilities of subsets of variables (building blocks). The name and motivation for the cGA come from the idea that members of a GA population might be modelled by an area of high probability (the attractors) in the MPM. This leads to the observation that there is a strong relationship between the capacity of a probabilistic model and the diversity of the population of a GA it is able to replace, which is one reason why it is important to study the capacity of EDAs.

The Bivariate Marginal Distribution Algorithm (BMDA) [14] is an example of a second order model. It does not build a fully connected model like the HEDA, but discovers a sparse set of second order dependencies between variable pairs. The sparse nature of the connections means that subsets of variables may be completely separate from the rest, meaning that a set of graphs (known as a forest) is produced. The more sparse the network, the lower its capacity for storing attractors. The connections in the BMDA are conditional probabilities, making the model more akin to a Bayesian network, whereas the HEDA structure is more like that of a Markov Random Field.

A recent example of a high order EDA can be found in the multi-variate DEUM model, [23]. DEUM performs distribution estimation using Markov random fields (MRF). The HEDA shares its structure with a second order MRF but differs in the way it represents the fitness function. An MRF attempts to model the probability distribution of highly fit patterns as a product across cliques in the graph, which is equivalent to a Gibbs distribution, in which the probability of a pattern across the inputs is calculated as the exponential of the energy state.

$$P(\mathbf{x}) = \frac{1}{Z} e^{-Tu(\mathbf{x})} \tag{30}$$

where u is the energy function, equivalent to Eq. 14 and Z is the normalising constant, which can be ignored in the context of network capacity as dividing the energy by a constant has no effect on the location or number of attractor

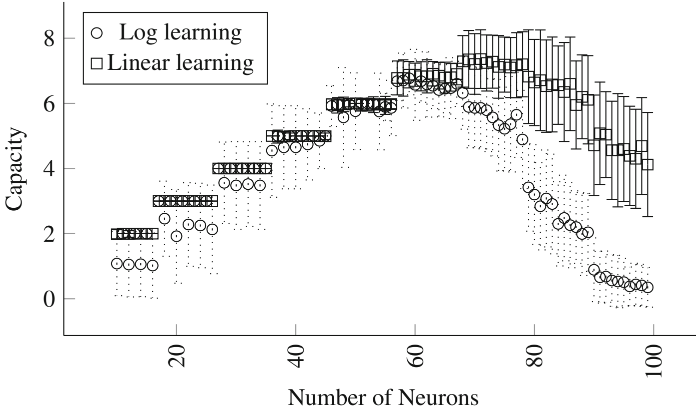

Fig. 9. Number of attractors correctly discovered from the same random set by a HEDA with a standard linear learning rule (square markers and solid error bars) and a HEDA trained with the log-Hebbian rule. In both cases, the target function contained $n(2\ln n)$ attractors and the graph shows how many of them were found.

states. When using the MRF as an EDA, new candidate solutions are sampled by a stochastic probability hill climb such as the Metropolis Hastings algorithm [24] or simulated annealing. Attractor points have an exponentially higher probability of appearing in a sample due to Eq. 30 so the number of attractors (and hence, the capacity of the network) is an important consideration. The similarity between a MRF based EDA and a HEDA is clear. The HEDA models the fitness function rather than the probabilities explicitly, but differs from the probability distribution in that the estimate of the fitness is the natural log of the estimate of the probability of a pattern. The HEDA can be made to learn the probability distribution in the same way as the MRF by changing the learning rule to become a log-Hebbian rule:

$$w_{ij} \leftarrow w_{ij} + \ln(f(\mathbf{c}))u_i u_j \tag{31}$$

Testing the log-Hebbian rule on the Hamming distance function revealed it to have a lower capacity than the standard HEDA rule, which is to be expected because the Hamming function is based on a sum rather than a product across variables. Figure 9 shows the results of testing the capacity of a number of differently sized HEDAs trained on both the Hebbian and the log-Hebbian rule. The models were tested on Hamming distance functions with m random attractors where m was set to be the theoretical capacity for the size of the network. In these experiments, capacity is defined as the number of attractors from the target list successfully found. As the networks pass 70 neurons in size (6 patterns) higher order interactions begin to damage stored memories and the number of patterns correctly stored starts to fall.

It might be argued that the HEDA is not an EDA at all, but a fitness function model as its energy function is a second order model of the fitness function. Its use in optimisation, however, is akin to that of an EDA as it can be used to sample

from promising areas of the input space by minimising its energy function. The HEDA can be used as a true EDA by learning the natural log of the fitness function rather than the function itself.

9 Conclusions and Further Work

Hopfield networks have previously been used as optimisation tools but the weights have always been designed by hand. The contributions of this paper are twofold. It presents a method for automatically discovering the weight values for a Hopfield network from samples from a fitness function and an analysis of the capacity of such networks for storing local optima as attractor points. An analysis of linkage order and network capacity has shown that such second order networks can learn all of the attractor states of some higher order functions, even when they cannot reproduce the function output reliably.

The attractors of a HEDA can be viewed as diverse members of a GA population. They are a set of current best points in the search. The capacity of the network limits the number of such points that can be stored at any one time and exceeding the capacity damages all the other existing memories held in the network. For this reason, some form of attractor management may be required to ensure that newly emerging attractors have higher scores than the ones they are destroying.

The next step in this research is to develop an evolutionary algorithm that can store an evolving set of attractor states and sample from them to produce a model of a small quantity of high fitness solutions. The role played by spurious states needs further investigation, as does the effect of adding higher order weights. [25] states that the capacity of order m associative memories over n neurons is $O(n^m/\ln n)$ but the trade-off between network capacity, network size and model overfitting needs careful management. Work on discovering useful heuristics for sampling the space of possible weights to optimise the multiple goals of model accuracy and small network size is ongoing, for example [26] describes some work on the use of higher order versions of the HEDA to learn and sample from distributions.

References

1. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley Professional, Upper Saddle River (1989)
2. Goldberg, D.E.: Genetic algorithms and walsh functions: Part II, deception and its analysis. Complex Syst. **3**, 153–171 (1989)
3. Holland, J.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. University of Michigan Press, Ann Arbor (1975)
4. Pelikan, M., Goldberg, D.E., Cantú-paz, E.E.: Linkage problem, distribution estimation, and bayesian networks. Evol. Comput. **8**(3), 311–340 (2000)
5. Davidor, Y.: Epistasis variance: a viewpoint on ga-hardness. In: Rawlins, G.J.E. (ed.) Foundations of Genetic Algorithms, pp. 23–35. Morgan Kaufmann, San Mateo (1990)

6. Goldberg, D.E.: Genetic algorithms and walsh functions: part I, a gentle introduction. Complex Syst. **3**, 129–152 (1989)
7. Bethke, D.: Genetic algorithms as function optimizers (1978)
8. McEliece, R., Posner, E., Rodemich, E., Venkatesh, S.: The capacity of the hopfield associative memory. IEEE Trans. Inf. Theory **33**(4), 461–482 (1987)
9. Santana, R.: Estimation of distribution algorithms: from available implementations to potential developments. In: Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation, pp. 679–686. ACM (2011)
10. Baluja, S., Caruana, R.: Removing the genetics from the standard genetic algorithm. In: ICML, 38–46. Morgan Kaufmann (1995)
11. Harik, G., Lobo, F., Goldberg, D.: The compact genetic algorithm. IEEE Trans. Evol. Comput. **3**(4), 287–297 (1999)
12. Mhlenbein, H.: The equation for response to selection and its use for prediction. Evol. Comput. **5**(3), 303–346 (1997)
13. Bonet, J.S.D., Isbell Jr., C.L., Viola, P.: Finding optima by estimating probability densities. In: Mozer, M., Jordan, M., Petsche, T. (eds.) Advances in Neural Information Processing Systems, p. 424. The MIT Press, Cambridge (1996)
14. Pelikan, M., Mühlenbein, H.: The bivariate marginal distribution algorithm. In: Roy, R., Furuhashi, T., Chawdhry, P.K. (eds.) Advances in Soft Computing - Engineering Design and Manufacturing, pp. 521–535. Springer, London (1999)
15. Pelikan, M., Goldberg, D., Cant-Paz, E.: Linkage problem, distribution estimation, and bayesian networks. Evol. Comput. **8**(3), 311–340 (2000)
16. Shakya, S., McCall, J., Brownlee, A., Owusu, G.: Deum - distribution estimation using markov networks. In: Shakya, S., Santana, R. (eds.) Markov Networks in Evolutionary Computation. Adaptation, Learning, and Optimization, vol. 14, pp. 55–71. Springer, Berlin (2012)
17. Walsh, J.: A closed set of normal orthogonal functions. Am. J. Math. **45**, 5–24 (1923)
18. Hopfield, J.J.: Neural networks and physical systems with emergent collective computational abilities. Proc. Nat. Acad. Sci. USA **79**(8), 2554–2558 (1982)
19. Hopfield, J.J., Tank, D.W.: Neural computation of decisions in optimization problems. Biol. Cybern. **52**, 141–152 (1985)
20. Caparrós, G.J., Ruiz, M.A.A., Hernández, F.S.: Hopfield neural networks for optimization: study of the different dynamics. Neurocomputing **43**(1–4), 219–237 (2002)
21. Ackley, D., Hinton, G., Sejnowski, T.: A learning algorithm for Boltzmann machines. Cogn. Sci. **9**(1), 147–169 (1985)
22. Storkey, A.J., Valabregue, R.: The basins of attraction of a new hopfield learning rule. Neural Netw. **12**(6), 869–876 (1999)
23. Shakya, S., Brownlee, A., McCall, J., Fournier, F., Owusu, G.: A fully multivariate deum algorithm. In: IEEE Congress on Evolutionary Computation, CEC '09, pp. 479–486 (2009)
24. Chib, S., Greenberg, E.: Understanding the metropolis-hastings algorithm. Am. Stat. **49**(4), 327–335 (1995)
25. Kubota, T.: A higher order associative memory with Mcculloch-Pitts neurons and plastic synapses. In: International Joint Conference on Neural Networks, IJCNN 2007, pp. 1982–1989 (2007)
26. Swingler, K., Smith, L.S.: Mixed order associative networks for function approximation, optimisation and sampling. In: Proceedings of 21st European Symposium on Artificial Neural Networks, ESANN 2013 (2013)