MIXED ORDER HYPER-NETWORKS FOR FUNCTION APPROXIMATION AND OPTIMISATION

KEVIN SWINGLER

Doctor of Philosophy

Computing Science and Mathematics
University of Stirling

May 2016

ABSTRACT

Many systems take inputs, which can be measured and sometimes controlled, and outputs, which can also be measured and which depend on the inputs. Taking numerous measurements from such systems produces data, which may be used to either model the system with the goal of predicting the output associated with a given input (function approximation, or regression) or of finding the input settings required to produce a desired output (optimisation, or search). Approximating or optimising a function is central to the field of computational intelligence.

There are many existing methods for performing regression and optimisation based on samples of data but they all have limitations. Multi layer perceptrons (MLPs) are universal approximators, but they suffer from the *black box* problem, which means their structure and the function they implement is opaque to the user. They also suffer from a propensity to become trapped in local minima or large plateaux in the error function during learning. A regression method with a structure that allows models to be compared, human knowledge to be extracted, optimisation searches to be guided and model complexity to be controlled is desirable. This thesis presents such as method.

This thesis presents a single framework for both regression and optimisation: the mixed order hyper network (MOHN). A MOHN implements a function $f : \{-1, 1\}^n \rightarrow \mathbb{R}$ to arbitrary precision. The structure of a MOHN makes the ways in which input variables interact to determine the function output explicit, which allows human insights and complexity control that are very difficult in neural networks with hidden units. The explicit structure representation also allows efficient algorithms for searching for an input pattern that leads to a desired output. A number of learning rules for estimating the weights based on a sample of data are presented along with a heuristic method for choosing which connections to include in a model. Several methods for searching a MOHN for inputs that lead to a desired output are compared.

Experiments compare a MOHN to an MLP on regression tasks. The MOHN is found to achieve a comparable level of accuracy to an MLP but suffers less from local minima in the error function and shows less variance across multiple training trials. It is also easier to interpret and combine from an ensemble. The trade-off between the fit of a model to its training data and that to an independent set of test data is shown to be easier to control in a MOHN than an MLP.

A MOHN is also compared to a number of existing optimisation methods including those using estimation of distribution algorithms, genetic algorithms and simulated annealing. The MOHN is able to find optimal solutions in far fewer function evaluations than these methods on tasks selected from the literature.

ACKNOWLEDGMENTS

I've been supported in writing this work by my wife Maxine who put up with my late nights in the office, especially during the final stages. My kids have little idea what Daddy does instead of coming home to feed them fish fingers, but their smiling little faces kept me going all the same. Thank you Sam and Jamie. My parents have been a great help, both now by looking after the kids and throughout my life by encouraging me to always learn and explore.

My supervisors, Prof. Leslie Smith and Prof. Amir Hussain deserve particular thanks and I'd further like to thank David Cairns, Sandy Brownlee, John McCall, Amos Storkey, Jerry Swan and Bruce Graham for various advice, discussions and help. In fact, I doubt that there is a single member of staff in the Computing Science department at Stirling who hasn't made this work a little bit better in one way or another.

PUBLICATIONS

This thesis has produced ten peer reviewed publications. There is also one further paper in preparation, which compares MOHNs to EDAs.

1. [125] Kevin Swingler, Leslie S. Smith. On the capacity of Hopfield neural networks as EDAs for solving combinatorial optimisation problems. *In Proc. IJCCI (ECTA)*, pages 152-157. 2012: Selected as invited paper for special issue journal.

2. [132] Kevin Swingler, Leslie S. Smith. Mixed order associative networks for function approximation, optimisation and sampling. *In Proc. ESANN 2013, 21st European Symposium on Artificial Neural Networks, Proceedings*: Selected as invited paper for special issue journal.

3. [134] Kevin Swingler, Leslie S. Smith. Training and making calculations with mixed order hyper-networks. *Neurocomputing*, (141) pages 65-75, 2014

4. [133] Kevin Swingler, Leslie S. Smith. An analysis of the local optima storage capacity of Hopfield network based fitness function models. *TCCI Special Issue*. 2014

5. [126] Kevin Swingler. A Walsh analysis of multilayer perceptron function. *In Proc. IJCCI (NCTA)*, 2014: Shortlisted for best student paper

6. [129] Kevin Swingler. Opening the Black Box: Analysing MLP Functionality Using Walsh Functions, *Studies in Computational Intelligence*, 2016

7. [128] Kevin Swingler. Local Optima Suppression Search in Mixed Order Hyper Networks, *Proc. UKCI*, 2015

8. [127] Kevin Swingler. A Comparison of Learning Rules for Mixed Order Hyper Networks, *In Proc. IJCCI (NCTA)*, 2015: Winner of best student paper

9. [130] Kevin Swingler. Structure Discovery in Mixed Order Hyper Networks, *Big Data Analytics*, 1. 2016

10. [131] Kevin Swingler. High Capacity Content Addressable Memory with Mixed Order Hyper Networks, *Studies in Computational Intelligence*, in press

These papers are available to download from www.mixedorder.net, which is a site created by the author as a resource for researchers interested in this field. Other outputs from the thesis are also available at the site, including the slides from conference talks, Java code for implementing a MOHN and its attendant methods and short descriptions of the methods and animations of the algorithms in action.

CONTENTS

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF SYMBOLS AND ABBREVIATIONS

The following conventions are used throughout this thesis. Random variables or vectors of random variables are denoted using upper case roman letters. Particular instantiations or realisations of a variable are denoted by its lower case equivalent. Sets are denoted with bold upper case roman letters and their members are represented with the same letter in lower case, with an index subscript. Indices are always lower case letters. Parameters, both statistical and control are denoted using Greek characters.

$X = X_1 \dots X_n$	The vector of inputs to a function
n	The number of inputs to the function (the size of X)
i	i is always used to index inputs (either input variables or their corresponding neurons)
$x = x_1 \dots x_n, x_i \in \{-1, 1\}$	Realisations of X
Y	The output from a function, e.g. $Y = f(X)$
y	A realisation of the output required from a function
$\mathbf{D} = D_1 \dots D_m$	The data used to build a function approximation is held in a multiset, $\mathbf{D}$ of size m
$x, (x, y) \in \mathbf{D}$	Each element in $\mathbf{D}$ is a (vector, scalar) pair, (x, y) or, in the case where the function to be learned is a content addressable memory, an input vector, x.
$\langle a \rangle$	Angled brackets denote an average (the mean, unless otherwise stated) or expected value.
$M = (X, \mathbf{W})$	Defines a MOHN with inputs X and a weight set $\mathbf{W}$
$W_j = (\omega_j, \mathbf{I})$	Each weight in $\mathbf{W_j}$ is a pair consisting of a scalar valued weight, ω_j and a set of neuron indices, $\mathbf{I}_j$ that define the connected neurons.
j	j is always used to index weights in a MOHN.
$f(X)$	A function that generates data from which a model of $f(X)$ will be built
$\hat{f}(X)$	An approximation to $f(X)$ made by a statistical model
$\hat{Y}$	The estimate of the value Y made by $\hat{f}(X)$

ABBREVIATIONS

MOHN	Mixed Order Hyper Network
MSDA	MOHN Structure Discovery Algorithm
SGD	Stochastic Gradient Descent
RMSE	Root mean squared error
OLS	Ordinary Least Squares
The lasso	The least absolute shrinkage and selection operator
LOSS	Local Optima Suppression Search
WSS	Weight Satisfaction Search
SA	Simulated Annealing
RRHC	Random Restart Hill Climb
ILS	Iterated Local Search
MLP	Multi Layer Perceptron
HN	Hopfield Network
MRF	Markov Random Field
DEUM	Distribution Estimation Using Markov random fields
GA	Generic Algorithm
BOA	Bayesian Optimisation Algorithm
hBOA	Hierarchical Bayesian Optimisation Algorithm
BBN	Bayesian Belief Network

Part I

INTRODUCTION

1

INTRODUCTION

1.1 Setting the Scene

Many real world systems can be characterised as a function that maps a number of inputs onto an associated output. The system might be a machine whose settings affect product quality, or the way the demographics of a customer affect how much they spend, or how the changes in a schedule affect its efficiency, or even the effectiveness of a strategy in a sports match. If data that measures the inputs and outputs can be collected, then its analysis may reveal useful insights about the system.

Statistical models of these functions help us to understand the system behind them, predict outcomes for known inputs and discover patterns of inputs that lead to desired outcomes. Sometimes a function that emulates the real world can be programmed, but it is often the case that the only resource available is a sample of data describing input, output pairs. Alternatively, a computer model of a function may be available but costly to evaluate. Two aspects of the field of computational intelligence are concerned with learning and searching functions. Machine learning (or statistical learning, or data mining) is concerned with the task of learning a function from a sample of data. This process is known as regression or, more generally, predictive analytics. Heuristic optimisation is concerned with finding input values that optimise a desired quality in the output of a function (usually minimising or maximising it). This is often known as prescriptive analytics.

Both regression and heuristic optimisation play an important role in many of the sciences, commerce, engineering, finance, medicine and sport. A large body of research proposing methods for carrying out these tasks has been generated and active research continues to investigate methods to improve regression models and find desired outputs using fewer evaluations of the function.

The main requirement of a model built from data is that it generalises well to data that it has not seen before. That requires the model to extract sufficient signal from the data while ignoring the noise. It is often the case that a model that performs very well on the data used to build it will generalise worse than one that is purposefully limited in its ability to learn from the data it has. A good model, then, is one which is powerful enough to capture the essence of the system that generated the data, but not so powerful that it captures noise and the effects of sampling.

Each input variable has an effect on the output. Taking any input pattern and changing the value of a single variable in it will cause a change in the output. If the size of the change in

output is always the same when a given variable is changed by a fixed amount, the relationship between that variable and the output is linear. If the size of the change in the output following a fixed change in one variable depends on the values of other variables, then the relationship is non-linear and we say that the variables involved *interact*.

Any model has more expressive power when it is able to take into account the interactions among the inputs as they contribute to the value of the output. If the system that produced the data involves interactions the model cannot capture, the model will be too simple. If the model captures interactions that are artifacts of the sample or of noise, it will generalise poorly. Similarly, a function in which few of the inputs interact to influence the output is generally easier to optimise than one where many such interactions are present. This thesis presents a method of performing regression that makes the interactions among input variables explicit. In the limited scope of functions that map many binary input variables onto a single real valued output, it presents a method that can represent any function to arbitrary precision, provides a human readable representation of the input interactions that contribute to the output, and provides some insight that can be used to find optimal input values.

1.2 Scope

The systems under consideration in this work are of n binary variables that map to a real valued output, i.e. $Y = f(X)$ where $X \in \{-1, 1\}^n$ and $Y \in \mathbb{R}$. A single model, the mixed order hyper network (MOHN) is proposed that may be used for regression (learning f), optimisation (finding a value of X that maximises Y) or building a content addressable memory (such that local maxima in f represent stored patterns).

1.2.1 *Notation*

The input vector is denoted by X and the scalar output is denoted by Y. The number of inputs in X is n, each one being denoted X_i $i = 1 \ldots n$. Models are built from a training data set of (input, output) pairs. Let m be the size of that data set. The input data are represented as an $m \times n$ matrix, $\mathbf{X}$. Rows in $\mathbf{X}$ represent single training examples and are denoted x_j $j = 1 \ldots m$. The set of output values associated with Y are denoted $\mathbf{Y}$ and their individual values are denoted y_j $j = 1 \ldots m$. A single training example is (x_j, y_j). The predicted value of Y from a statistical model is $\hat{Y}$ and a model of f, built from samples (x, y) is denoted $\hat{f}$.

The assumption is made that the data are generated from a system in the real world with an underlying function of its own. Let this function be called the *target function*, denoted f. The general assumption is that the data are generated by a mixture of the source function and

additional unexplained noise so $Y = f(X) + \epsilon$ where ϵ has a mean of zero. The model is noise free, so $\hat{Y} = \hat{f}(X)$.

1.3 Thesis

This thesis presents mixed order hyper networks (MOHNs) as regression models in $f : \{-1, 1\}^n \rightarrow \mathbb{R}$. It also addresses the use of MOHNs as fitness function models in optimisation tasks. For regression modelling, the primary concern is finding the correct level of bias. For optimisation the focus is on the trade-off between the quality of the solution and the time (or number of fitness evaluations) required to find it. MOHNs are shown to have the following properties:

1. **Basis Function**: MOHNs form a basis $f : \{-1, 1\}^n \rightarrow \mathbb{R}$, meaning that all functions in that domain can be modelled by a MOHN. This means that any MOHN may have zero model bias.

2. **Sparsity**: There are functions whose basis representation contains many zero valued parameters, meaning that such functions may be learned by models with fewer parameters than are required to define the full basis.

3. **Linear Parameter Models**: MOHNs are linear parameter models, having all the properties associated with such models. It has the ability to learn a function from a number of noise free data points equal in size to the number of parameters in the model, and there are convex cost functions available for estimating the parameter values;

4. **Interpretability**: The structure and values of the parameters in a MOHN have a meaning that is open to human interpretation. This allows networks to be visualised, compared and (to some extent) produce human readable facts.

Heuristic algorithms for performing two important tasks with MOHNs are proposed:

1. **Structure Discovery**: An algorithm designed to discover the non-zero parameters in a MOHN is presented. The algorithm attempts to balance the trade-offs among the number of data points required, the scope of the parameter search and the speed of learning.

2. **Model Search**: Algorithms that attempt to make use of the structure of the MOHN to guide local search are presented and tested.

The thesis also presents experimental evidence to suggest that

1. **Non-Linear Regression**: For a number of benchmark functions, a MOHN showed advantages over a multi layer perceptron including finding a lower test error, requiring

fewer data points, using fewer training epochs and displaying less variance across a number of training runs.

2. **Fitness Function Models**: MOHNs are capable of modelling benchmark fitness functions and finding the input values that produce the global maximum output of those functions in fewer evaluations of the fitness function than a number of published state-of-the-art methods.

These claims are listed again in section 5.2.1 with references to parts of the thesis that prove or demonstrate them.

1.4 Plan of the Thesis

The rest of this thesis is organised as follows. Chapter 2 reviews the literature on function modelling, optimisation and graphical models. Chapter 3 introduces mixed order hyper networks and describes methods for building, training and searching them. Chapter 4 describes a set experiments designed to test and demonstrate the use of a MOHN. Section 4.9 provides a case study based on data from a mail order clothing company and section 4.11 compares the use of a MOHN to a number of other heuristic optimisation methods on problems from the recent literature. Finally, chapter 5 summarises the work and proposes some further research directions.

2

LITERATURE REVIEW

2.1 Existing Work

A large body of research has addressed the three questions of regression, heuristic optimisation and content addressable memories. This section describes those that are considered most relevant to the methods proposed in this thesis and also summarises some of the key concepts on which they are based.

2.1.1 *Statistical Learning*

Statistical learning is the process of using data to fit the parameters of a statistical model so that it displays a desired behaviour. Generally the desired behaviour is to reflect some statistical properties of the data and is defined in terms of minimising a cost function with respect to those data. In general, the class and structure of the model is chosen and fixed before the parameters are fitted but in some cases, parameters are added or removed dynamically during the process of learning.

2.1.1.1 *Cost Function Minimisation*

Statistical models are generally built with reference to a cost function and the job of a learning algorithm is to find a set of parameters for the model that minimise that cost. Common cost functions measure the error of a model or its likelihood and some cost functions have penalty terms to control complexity. In some cases, the cost function may be minimised analytically by setting the derivative to zero and solving the resulting set of equations across the training data. In other cases, this is not possible and an iterative approach is required, either descending the gradient of the cost function locally or by other heuristics. In such cases, the partial derivative of the cost function, C with respect to a given parameter, ω, $\frac{\partial C}{\partial \omega}$ is calculated to guide parameter changes.

A common cost function is the mean squared error (MSE), which is the average of the squared distance between model output and measured output in the data. The average is taken across a data set, so the same model may have different MSE values for different data sets (training data

and test data, for example). Given a data set with m inputs, x_j and associated outputs, y_j for $j = 1 \dots m$, MSE is calculated as

$$MSE = \sum_{j=1}^{m} (y_j - \hat{f}(x_j))^2 \tag{2.1}$$

2.1.1.2 *The Bias, Variance Trade-Off*

If the job of a predictive model were to simply minimise a cost function that measured the distance between the model and each data point in the training set, this could be achieved by simply using the data as a look-up table. A model has the advantage of being smaller and faster than such a lookup, but the main advantage of a model is that has the ability to *generalise*, producing outputs for input values that were not in the training data.

The training data is a sample from all the possible data (the population) that could be measured and modelled and any statistical properties that are estimated from it will have some degree of sampling error. Noise and sampling variation in the data mean that different models built on different samples have the potential to differ from each other. As sample size grows, this variation is reduced, but the curse of dimensionality means that the required sample size grows exponentially with the number of inputs. In high dimensions, training data becomes sparse and the effect of variation in training samples increases.

A model that learns the training data too specifically is likely to generalise worse than a simpler model that learns a more parsimonious representation of the data. This is known as over fitting. Conversely, a model that is too simple may perform poorly on both training and test data. This is known as under fitting. These concepts contribute to a trade-off between two qualities of a model known as bias and variance.

Bias and variance both measure the expected value of different contributions to model error over many different samples of the population. Assume that there is an unknown target function underlying the data, $Y = f(X)$ and that there is natural variance around the expected value of Y given any X so that the data satisfies $Y = f(X) + \epsilon$ where $\epsilon \sim N(0, \sigma)$ accounts for that variation. Any given $\hat{y} = \hat{f}(x)$ is an estimate of the mean of the output given an input of x. Across the sampling distribution of a particular model, the expected value of $\hat{Y}$ will differ from the true value of Y by an amount known as the bias. Bias is defined as

$$E[\hat{f}(x) - f(x)] \tag{2.2}$$

Bias can be further decomposed into model bias and estimation bias. Model bias is error that is due to the model form and estimation bias is error that is due to the parameter values. A model can have low estimation bias (its parameters are correctly estimated) but high model bias (the model is a poor choice for the data). Models with zero estimation bias, such as the

results of using ordinary least squares are said to be unbiased. Section 2.1.3 will discuss biased and unbiased linear approximators.

The sampling distribution of models over which the bias is the expected value of the error has an associated variance,

$$E[\hat{f}(x) - E[\hat{f}(x)]]^2 \tag{2.3}$$

which measures the variation in output values from models built across all the different possible samples. The mean squared error (MSE) can be decomposed into a sum of ϵ, squared bias and variance. There is a trade-off between how bias and variance contribute to MSE. Bias can be reduced by fitting a more complex model, but at the expense of increased variance or, alternatively, variance can be reduced by simplifying or regularising a model at the expense of an increase in bias.

2.1.1.3 *Regularisation*

The bias, variance trade-off is managed by controlling the complexity of the model learned from the data, a process known as regularisation. Regularisation introduces bias to a statistical model in an attempt to reduce over fitting. Estimation bias can be introduced by ensuring that the squared error cost function is not completely minimised and model bias can be introduced by reducing the expressive power of the model, usually be removing parameters.

A statistical model is characterised by a set of parameters and an algorithm for using those parameters to map an input vector to an output. The algorithm requires the parameters to have a given structure. For example, linear regression requires a single parameter per input variable, multi-layer perceptrons expect parameters describing weights connecting one or more layers of hidden units, and a regression tree algorithm expects a set of branching decisions based on the input variables. These methods are described in more detail below, but for now, we note that different models have different restrictions, and so different model bias.

Some types of model have a fixed number of parameters (linear regression on a fixed number of inputs is an example) but others, such as neural networks, allow a variable number of parameters to be used. In general, adding more parameters produces a more complex model and should be expected to reduce bias (up to a point) but increase variance. Restricting the parameters' magnitudes can reduce variance, but introduces estimation bias. The L_1 norm measures the sum of the absolute values of the parameters, for example and is used as a regularisation term in the lasso (least absolute shrinkage and selection operator) [136]. The L_2 norm measures the Euclidean length of the parameter vector and is used for regularisation in ridge regression [61].

Other methods may also be used to achieve some form of regularisation. For example, methods that iterate over a data set many times to reduce error can keep track of both train and

validation error and stop when the validation error starts to rise consistently. This is known as early stopping. Alternatively, the addition of noise or other alterations to the training data can reduce the risk of over fitting.

The discussion above highlights the notions of model complexity and target function complexity. Part of the task of balancing the bias/variance trade-off involves finding a degree of complexity for a model that matches that of the target function. That process is made explicit by this work.

2.1.2 *Variable Selection*

In addition to choosing a model with the right bias and a correctly regularised set of parameter values, it is also necessary to make choices about the input variables to include in a model. Reasons to reduce the number of variables used include the desire for a simpler, parsimonious model and the requirement for a data set that is larger than the number of parameters in a model.

Approaches to feature selection can be separated into two classes. Embedded methods build a model using all of the variables and use its resulting structure to inform the process of choosing those to remove. Where model structure is difficult to interpret, feature selection may be done in an independent pre-modelling step. This approach may be incorporated into an iterative search in which feature selection and model building are alternated in an attempt to find the right variables for a given modelling technique. These are known as *wrapper methods*, [78].

There are a number of greedy methods for adding (and sometimes removing) variables one at a time in an attempt to reduce the minimised cost function of the resulting model. Stepwise regression [60] adds variables one at a time based on their correlation with the output and removes any that are rendered insignificant in an F-test as a result of recently added variables. Hall [48] proposed a similar correlation based feature selection method, which includes variables that are correlated with the output but removes variables that are correlated with those already selected for inclusion in the model.

The max-dependency, max-relevance, and min-redundancy (mRMR) approach [105] to feature selection uses a measure of mutual information between the input variables and the output class alongside measures of mutual information between different input variables to attempt to maximise the dependency of the output on the input while minimising the shared information (i.e. redundancy) between inputs.

This thesis concentrates on frequentist approaches but there are Bayesian approaches to feature selection that should also be mentioned. The Bayesian approach treats each parameter in the model as a random variable subject to a prior. The training data is used to infer the posterior distribution over each variable. For feature selection, the choice of prior is made in an attempt to reflect the assumption that many of the parameters have a value of zero. Selecting

a Gaussian prior over parameter values is equivalent to performing L_2 regularisation and a Laplace prior is equivalent to the L_1 regularisation method the lasso. See 2.1.3.3 for more details on these regularisation methods. A popular prior for Bayesian variable selection, which is similar to an L_0 regularisation (which is a count of the number of non-zero parameters) is the so-called spike and slab prior [68]. The spike and slab prior is defined as having very low, uniform probability at all values except zero (the slab) and a high probability at (or around) zero (the spike) [97]. The prior probability distribution over all possible models is then given as the product of the spike probability over the excluded parameters times the product of the slab probability over those that are included. Having defined a prior, the posterior, which is the probability of the model given the data is calculated using the standard Bayesian approach of multiplying the prior by the probability of the data, given the model (the likelihood).

Another Bayesian approach of note is automatic relevance determination (ARD), sometimes called sparse Bayesian learning [92] [100]. The ARD approach attempts to minimise a cost function that includes a regularisation term that can vary across the model parameters. This can be achieved using a Gaussian prior with zero mean and a precision term (the reciprocal of the variance) for each model parameter. Learning involves discovering the precision hyperparameters associated with each model parameter. High precision indicates high confidence that the parameter value is at or close to zero and might be excluded.

Greedy methods all suffer when variables are uninformative in isolation but have predictive power in combination. Take the XOR function as a simple example: neither input is correlated with the output in isolation but together they define the function space perfectly. To solve this problem, a method is needed that can consider variable subsets atomically. Genetic algorithms (GAs) have some ability to solve this problem and have been used to search for optimal feature sets by a number of authors. For example, Bala et al. [7] use a hybrid GA and wrapper approach to feature selection. Cantú-Paz [23] compared GAs with three other evolutionary computing methods, namely Estimation of Distribution algorithms (EDAs), Compact GAs, and Bayesian Optimisation Algorithms (BOAs) in terms of their ability to perform feature selection.

Ideally, then, feature selection should be seen as integral to the data modelling process and part of model bias control, a question not just of which variables to include, but which interactions to model. A representation that makes those interactions explicit and an algorithm for efficiently exploring which to include is desirable for those reasons.

2.1.3 *Regression Methods*

The regression methods described here are not restricted to binary valued input variables, but can be used in that space.

2.1.3.1 *Multiple Linear Regression*

Multiple linear regression assumes a linear relationship between a vector, X and a scalar, Y. It assumes that each variable, X_i in X has an influence on Y that is independent of any other variable in X. The linear model predicts the expected value of Y at each point in X.

REPRESENTATION A linear combination of X is used to predict the expected value of Y

$$\hat{Y} = \beta_0 + \sum_{i=1}^{n} X_i \beta_i \tag{2.4}$$

where the β parameters define the independent contribution of each variable in X. For simplicity of notation, let $X_0 = 1$ and use vector notation so

$$\hat{Y} = X.\beta \tag{2.5}$$

The vector X can be the values of the input variables themselves or a new set of feature variables derived from the inputs. For example, if the input variables to be modelled are $V_1 \ldots V_p$, then a coefficient could be calculated for every product $V_i V_j$ to allow the model to take pairwise interactions into account. In theory, every interaction among variable subsets of all sizes up to n could be modelled but there are 2^n such interactions, so in practice the way that variables are combined to create the input features needs to be managed. Small feature sets are desirable for reasons of parsimony, efficiency and due to limitations imposed by small data sets. How variables are combined to form features is a key topic of this thesis.

LEARNING ALGORITHM There are a number of learning algorithms for multiple linear regression models. The most common is ordinary least squares (OLS), which is defined by its cost function

$$C = \frac{1}{2} \sum_{j=1}^{m} (y_j - \hat{f}(x_j))^2 \tag{2.6}$$

Let $\mathbf{X}$ be the $m \times (n+1)$ matrix of training data inputs and $\mathbf{Y}$ be the m-vector of target outputs, then from equation 2.5,

$$C = \frac{1}{2} (\mathbf{Y} - \mathbf{X}\beta)^T (\mathbf{Y} - \mathbf{X}\beta) \tag{2.7}$$

and C is minimised where the derivative, $\frac{\partial C}{\partial \beta_i} = 0 \; \forall i$. Solving this gives a least squares estimate for β of

$$\beta = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{Y} \tag{2.8}$$

BIAS AND COMPLEXITY Least squares is an unbiased estimator for the linear coefficients [52], that is to say that there is no estimation bias. Computing the OLS coefficients using singular value decomposition has a complexity of $O(mn^2)$ assuming that $m > n$.

2.1.3.2 *Stochastic Gradient Descent*

Stochastic Gradient Descent (SGD) [152] is an iterative approach to regression learning, which descends the error function in small steps in response to one training sample at a time, rather than building a matrix like OLS. Each step is guided by the derivative of the error function, a learning rate that restricts the size of the step and an optional regularisation term. Regularisation terms are considered in section 2.1.3.3 and this section considers the simpler case with no regularisation.

At each step, a single observation (x, y) is taken from the data and a predicted output, $\hat{f}(x)$ is calculated using the current set of parameter values, β. The change an individual parameter, β_i, is calculated from the derivative of the cost function, $C(\hat{f}(x), y)$:

$$\beta_i \leftarrow \beta_i - \eta_t \frac{d}{d\beta_i} C(\hat{f}(x), y) \tag{2.9}$$

where $0 < \eta_t < 1$ is a learning rate that can either be fixed to a constant value or reduced over time. When the cost function is the least squares, $C(\hat{f}(x), y) = \frac{1}{2}(\hat{f}(x) - y)^2$, the weight update becomes

$$\beta_i \leftarrow \beta_i - \eta_t (\hat{f}(x) - y) x_i \tag{2.10}$$

SGD is discussed in more detail in the context of training neural networks in section 2.1.4.

BIAS AND COMPLEXITY SGD with the least squares cost function asymptotically approaches the same result as OLS, but can be regularised by early stopping. It is also possible to add a regularisation term to the cost function, which is discussed next. SGD has a complexity of $O(mnp)$ where p is number of passes through the data. As m grows, p may be made smaller. SGD has the advantage when data sets are large that a small number of passes through the data are required and, in extremely large data sets, it may be possible to stop early before all of the data has been processed once (assuming the ordering of the data is not important). This can make SGD more efficient than OLS for large data sets in terms of time and memory [13].

2.1.3.3 *Shrinkage Methods*

Bias can be introduced into linear models using shrinkage methods, which impose a penalty on the size of the weights. This penalty is expressed as part of the cost function. For example, ridge regression [83] minimises

$$C = \sum_{k=1}^{m} (y_k - \hat{f}(x_k, \beta))^2 + \lambda \sum_{i=1}^{n} \beta_i^2 \tag{2.11}$$

where $\lambda \geq 0$ controls the amount of shrinkage. The ridge regression solution can be found at

$$\beta = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{Y} \tag{2.12}$$

where $\mathbf{I}$ is the $n \times n$ identity matrix.

Another popular regularised learning rule is the lasso (least absolute shrinkage and selection operator [136]), which aims to minimise the cost function

$$\frac{1}{2} \sum_{j=1}^{m} (y_j - \hat{f}(X_j, \beta))^2 + \lambda \sum_{i=1}^{n} |\beta_i| \tag{2.13}$$

where λ controls the degree of regularisation. When $\lambda = 0$, the lasso solution becomes the OLS solution. With $\lambda > 0$ the regularisation causes the sum of the absolute weight values to shrink such that weights with the least contribution to error reduction can take a value of zero.

The lasso weights cannot be found analytically as equation 2.13 cannot be differentiated, but a method called least angle regression (LARS) [41] can be used to efficiently calculate the lasso coefficients across the range of λ values. LARS takes a similar approach to forward stepwise regression, but adds variables in a way that is not as "all or nothing“. As each new variable is added, the model is moved towards the least squares fit of the selected variables and the model residual. At the point where an unused variable is as correlated with the residual as the current model, that new variable is added and the process continues.

The lasso implementation used throughout this thesis is Lasso4j [44], which is a Java library based on a cyclical coordinate descent approach [43]. This approach computes solutions along a path of values for λ, which is very efficient as it is able to make use of *warm restarts*. Once a coefficient reaches zero, it need no longer be considered further down the path, making greater speed ups possible.

BIAS AND VARIANCE Ordinary least squares regression is an unbiased estimator, but the lasso and ridge regression introduce estimation bias. The degree of bias can be controlled by the λ parameter in equations 2.11 and 2.13, which restricts the size of the coefficients and, in the case of the lasso, has the effect of causing some coefficients to go to zero. Model bias can be

controlled by the choice of which variables are combined to form the features, X that form the inputs to the model. First order multiple linear regression ignores possible interactions among variables, treating them as independent.

STRENGTHS AND LIMITATIONS First order MLR is simple to apply but has limited representational power. It assumes an independent linear relationship between each input and the output so the effect of interactions between input variables on the output are ignored.

FEATURE SELECTION For OLS, stepwise regression can be used for feature selection, as can any pre-modelling method. The lasso has the benefit of forcing some coefficients to zero, providing a built in feature selection mechanism. Parameters in a model trained with OLS can be tested for significance by calculating a Z-score

$$z = \frac{\hat{\beta}_j}{\hat{\sigma}\sqrt{v_j}} \tag{2.14}$$

where $\hat{\sigma}$ is the variance

$$\hat{\sigma} = \frac{1}{m-n-1}\sum_{j=1}^{m}(y_j - \hat{y}_j)^2 \tag{2.15}$$

and v_j is the jth diagonal element of $(\mathbf{X}^T\mathbf{X})^{-1}$. z_j follows a t distribution with $m-n-1$ degrees of freedom, which allows the null hypothesis that $\beta_j = 0$ to be tested.

2.1.3.4 *Generalised Linear Models*

Multiple linear regression assumes a normal distribution of errors across the output values associated with a given input and a linear relationship between the inputs and the output value. Both of these restrictions are relaxed in a generalised linear model (GLM), which allows the output errors to take any distribution from the exponential family. For example, logistic regression is a method for mapping input vectors onto class data (a Bernoulli distribution in the two class case).

REPRESENTATION GLMs take the form of a linear function and a non-linear link function that maps the linear output to the expected value of the desired distribution. The mean of the output distribution depends on X through the link function g:

$$Y = g^{-1}(f(X)) \tag{2.16}$$

where $f(X)$ is a linear combination and g^{-1} is the inverse of g.

In the class of link functions including those where the output distribution is Bernoulli, Binomial, categorical or multinomial, the output of the regression model is interpreted as a probability, $P(Y)$ and the link function is

$$X.\beta = \ln\left(\frac{P(Y|X)}{1-P(Y|X)}\right) \tag{2.17}$$

so

$$\frac{P(Y|X)}{1-P(Y|X)} = e^{X.\beta} \tag{2.18}$$

so

$$P(Y|X) = \frac{e^{X.\beta}}{1+e^{X.\beta}} = \frac{1}{1+e^{-X.\beta}} \tag{2.19}$$

which is the logistic function, and which will be returned to in the context of neural networks and Markov Random fields in sections 4.3.1, 2.3.1.3 and 2.3.1.6.

LEARNING ALGORITHM The most common learning algorithm for GLMs is iteratively reweighted least squares [63]. Regularisation can be imposed, for example using ridge regression [83].

2.1.4 *Multi Layer Perceptrons*

Feed forward neural networks do not assume that input variables have independent effects on the outputs. They can model the way interactions between input variables influence the output. This is achieved by chaining functions in a feed forward process through what are known as hidden layers. The hidden layers encode features of the data as non-linear functions of weighted sums of either the input variables or existing features from lower down the chain. A common feedforward neural network is the multilayer perceptron (MLP) [53].

REPRESENTATION Figure 2.1 shows the structure of an MLP with one hidden layer. Each node represents a function. Those at the bottom are the input nodes, which take a single input value and output that same value, unaltered. The other nodes receive a weighted sum of the outputs from connected nodes below, pass them through a function known as the activation function, and output a single value, which is passed as one of the inputs to every connected node in the layer above. At the output layer, this forms the output of the function. All layers, including the output can contain more than one node, allowing mappings from and to many variables to be learned. All neurons except those on the input also receive a constant input, known as the bias, on a weighted connection.

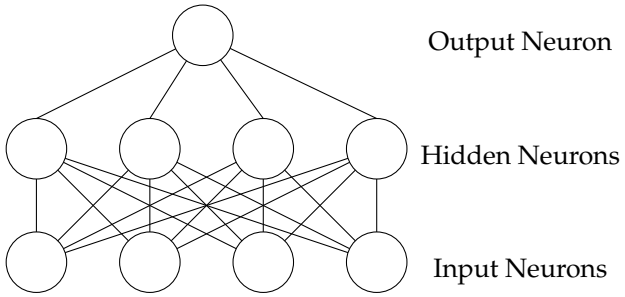

Figure 2.1: A multilayer perceptron with four input neurons, four hidden neurons and one output neuron. Bias weights are not shown.

Each neuron receives an activation, a_i, which is the sum of the products of the output from neurons below and the weights with which they are connected:

$$a_i = \sum_{l \in \mathbf{L}} w_{l,i} O_l \tag{2.20}$$

where $\mathbf{L}$ is the set of nodes in the layer below the one containing node i, $O_l = act_l(a_l)$ is the output from node l and $w_{l,i}$ is the weight of the connection from node l to node i. $\mathbf{L}$ consists of neurons that are either inputs (in the case of the first hidden layer) or neurons in the previous layer plus a single bias neuron with an output set permanently to one.

The activation function associated with neuron i, $act_i(a_i)$ may be linear, where $act_i(a_i) = a_i$ or non-linear depending on which layer it occupies and other design considerations that are touched on below. Common non-linear functions include the following:

The logistic function, whose choice is inspired by the output from logistic regression, described in section 2.1.3.4, where the output of node i is

$$act_i(a_i) = \frac{1}{1 + e^{-a_i}} \tag{2.21}$$

The tanh, which has a very similar shape to the logistic but is symetrical around zero (which the logistic is not), where the output of node i is

$$act_i(a_i) = \tanh(a_i) \tag{2.22}$$

The rectified linear function is linear for $a_i > 0$ but returns zero when $a_i < 0$ and is calculated by

$$act_i(a_i) = \max(0, a_i) \tag{2.23}$$

Neurons with a rectified linear activation function are called rectified linear units (ReLU). They do not suffer from the vanishing gradients of the sigmoid and tanh functions and are simpler (and so faster) to compute. When used in a hidden layer, ReLUs create a sparse representation as units with activation below zero have an output of zero and a gradient of zero, so their weights are not changed during learning. There is also a leaky ReLU, $act_i(a_i) = \max(\epsilon, a_i)$ where ϵ is small and positive (say 0.01), which can be used when a zero gradient needs to be avoided for negative activations.

In principle, each node could have a different activation function, but in practice, the inputs have a linear function, the hidden units all have the same non-linear function and the outputs have either a linear or non-linear activation function. The output of an MLP when the input is x, is defined as the output of the activation function on its output neurons. In the case of a single output, as discussed in this thesis, that is

$$\hat{y} = act_l(a_l) \tag{2.24}$$

where a_l is the activation of the single neuron at the output layer and $act_l()$ is the activation function associated with that neuron.

LEARNING ALGORITHM Most commonly, an MLP has a fixed structure of weights, the values of which are learned by a gradient descent of a cost function. A common cost function for performing regression is the quadratic.

$$C = \frac{1}{2m}\sum_{j=1}^{m}(y_j - \hat{y}_j)^2 \tag{2.25}$$

where $\hat{y}_j$ is the MLP output calculated using equations 2.24 and 2.20 in response to the input x_j. In order to minimise C, the partial derivative of the cost function with respect to each weight, $dC/dw_{l,i}$ needs to be calculated. This can be done one training example at a time. For a single training data point, the derivative of the cost function with respect to the output from the network is $dC/d\hat{y} = (y - \hat{y})$. The derivative of the error with respect to the activation, a of the output unit is dC/da, and depends on the choice of activation function. Each weight contributes $w_{l,i}O_l$ to a so the derivative $dC/dw_{l,i} = (y - \hat{y})dC/da(w_{l,i}O_l)$. Each weight is changed by $\eta dC/dw_{l,i}$ where $0 < \eta < 1$ is a learning rate that ensures individual weight changes are small. Errors are passed back through the network using the chain rule to allow earlier weights to be updated in a process known as back propagation of error [111].

NETWORK AND TRAINING DESIGN There are a number of design decisions that need to be made when building and training an MLP. Some concern the network itself and some concern the gradient descent learning algorithm. The decisions to be made are often framed as

a set of hyperparameters, which form a space that needs to be searched. Different choices of hyperparemeter values can lead to different functions being learned by the MLP and different levels of error. Commonly considered hyperparameters include the following, which are largely taken from a paper describing deep neural architectures [13]. Those listed here apply equally to MLPs.

NUMBER OF HIDDEN LAYERS AND UNITS MLPs can support many hidden layers but in reality only generally contain a small number. In principle a single layer containing a finite number of sigmoidal units and a linear output unit is sufficient to allow an MLP to approximate any continuous function on compact subsets of $\mathbb{R}^n$ [34]. However, the theory says nothing about the learnability of the weights and it has been found that adding several smaller hidden layers can be more efficient than training one large one [27]. The number of hidden units included in a network controls model bias. Too few neurons may make the network unable to represent the desired function and too many may lead to over fitting. Adding neurons can also increase training time as the number of weights to be updated increases.

ACTIVATION FUNCTION Some of the choices for activation functions are listed above. Hidden units should be non-linear. When performing regression, a linear output is used because the squared error cost function of equation 2.6 combined with a linear output corresponds to a Gaussian output model. For classification, a sigmoid activation function should be chosen at the output layer.

LEARNING RATE Gradient descent learning involves making small adjustments to model parameters (weights in the case of the MLP) to make steps down the error gradient. Making those steps too large causes the error rate to rise and making them too small causes the error to drop very slowly. The size of the changes in the weights is controlled by a learning rate. A good rule is that the learning rate should be the largest possible that does not cause the error to rise [13]. Variable learning rates are often used, with the learning rate diminishing according to a chosen schedule or in response to a flattening of the error rate.

EARLY STOPPING An easy method to avoid over fitting with an iterative learning process such as SGD is to terminate the training process before the training error has flattened. The error on a separate validation data set, which is not used to learn the model parameters, can be monitored so that early stopping can take place when the validation error begins to rise. Early stopping can obscure the over fitting effects of other hyperparameter choices, and so is best left out of an initial hyper-parameter search, and then used to attempt to improve a chosen configuration.

MOMENTUM In addition to a learning rate, weight updates are often smoothed using a moving average of previous updates. The proportions of the current gradient and of the previous average gradient (known as the momentum rate) that contribute to a weight change produce another hyperparameter that can be explored.

TRAINING BATCH SIZE Basic stochastic gradient descent makes one weight update per training example, approximating the change across all of the data by making small steps one at a time. Another alternative is to calculate the average gradient across all of the training data in a batch and make a single update to each weight based on a full pass through the data. The batch method makes the steps smoother but can be very slow as each weight update requires a complete pass through the training set. A compromise, known as mini batch training, updates the weights using the average error over small batches of training data. This smooths the error descent without slowing the process down to the same extent as a full batch approach.

WEIGHT INITIALISATION The weights of a MLP cannot be all set to zero before training begins. Their values must be randomised to avoid different neurons sharing the same weight values. Weights are often picked from a uniform distribution bounded by some range, the size of the range being one hyperparameter to explore. A recommended range is between $-r$ and r where $r = c\sqrt{6/(fanin + fanout)}$ and c is 1 for tanh functions and 4 for sigmoid functions [13]. The $fanin$ and $fanout$ values are the number of weights in and out of the unit associated with the weights being set.

DATA PREPROCESSING There are many options for pre-processing data in preparation for training an MLP. The only ones considered here involve shifting and scaling. For networks with sigmoid or tanh activation functions on their outputs, the output values must be scaled to the appropriate range (which is $[0, 1]$ for the logistic and $[-1, 1]$ for the tanh). It is also beneficial to standardise the input variables to a mean of zero and standard deviation of 1.

REGULARISATION Early stopping is mentioned above as one method for avoiding over fitting. Other regularisation methods applied to neural networks include the addition of noise to training data [14], penalties on weight size such as L1 and L2 regularisation and network structure methods such as dropout. Dropout [121] involves randomly ignoring a proportion of the neurons (and connected weights) in a network during training but using all of the neurons when testing. More specifically, during training each neuron is ignored with a certain probability, p, which attempts to turn a single network into the average of many sparse networks. When training is complete, weights are adjusted by a factor of p so that the output at test time is the same as the expected output during training.

OPTIMISING THE HYPERPARAMETERS Finding the right set of hyperparameters can have a large impact on the quality of an MLP. A simple method for searching for a good hyperparameter set is to perform a grid search. This involves defining a set of possible discrete values for each parameter and trying every combination. Each combination of hyperparameters is used to train at least one model and the error on a validation set is used to select the best combination.

STRENGTHS AND LIMITATIONS Multi layer perceptrons have the capacity to act as universal approximators. In reality, the correct architecture (the number of hidden units and the connectivity pattern among them) to represent a given function needs to be discovered and a learning algorithm must find the correct parameter settings. The cost function may contain local minima in which gradient descent methods may become trapped, making the testing of a chosen architecture more difficult.

This strength is accompanied by an increased risk (compared to simpler methods) of over fitting. Particular care is needed when training a neural network to achieve the correct trade-off between bias and variance. This task can be challenging due to two other well known weaknesses of MLPs.

The so called *black box* problem refers to the fact that the weights of an MLP offer very little in terms of human interpretability. It is not easy to extract information about the structure or complexity of the function a network has implemented. A lot of work has been dedicated to extracting rules or insights from multilayer perceptrons [4], [5], [67], [113]. Swingler [129] used a Walsh decomposition to investigate the structure of MLPs and showed that the complexity of the function being implemented by an MLP varied widely as it learned, demonstrating that simple measures such as the number of hidden units are, at best, a crude indication of function complexity.

Another well known problem with MLPs is the fact that the cost function contains local minima or plateaux from which a gradient descent algorithm cannot escape. Training a number of MLPs from different random starting weights can (if the error function dictates it) result in a number of different solutions. The black box problem described above compounds this problem as it is difficult to compare one MLP with another in terms of the structure of the function it implements. Swingler [129] presents evidence that the back propagation learning algorithm becomes trapped in local optima when its weights do not support the structural complexity required to model a function. The weights play two roles, firstly to encode the features represented by the hidden layer and secondly to map those features to the correct output values. Swingler suggests that networks in local optima encode the wrong features, meaning no correct mapping may be learned. This is demonstrated by showing MLPs trained on the XOR function (which is known to contain local minima) fail to encode the second order relationship among two inputs and the output required to model XOR. Once a network has

settled on a first order only feature representation, it cannot escape to the correct second order structure.

2.1.5 *Training, Testing and Validation*

Section 2.1.1.2 described how achieving good generalisation and avoiding over fitting are at the heart of statistical learning. An important question to be able to answer about a statistical model is "How well will it generalise to unseen data?". To answer the question, a subset of available data is kept aside for testing. This data must play no role in the selection of the model, the choice of training hyperparameters, or the model parameter setting. A single test set provides a single estimate of test error. A better estimate can be obtained by repeatedly training different models on different subsets of the data. A common method for balancing the bias-variance trade-off when training an MLP is to use cross-validation in which the data is partitioned into a number (usually ten, which is the figure used in the rest of this description) of non-overlapping test sets, each comprised of a different 10% of the data. Ten models are then trained, each on the 90% that remains for each test partition. The mean and variance of the error across these models gives a better indication of the likely error on new, unseen data.

When choosing a model and setting the training algorithm's hyperparameters, further division into training and validation data is required. For each combination of hyperparameters, a model needs to be trained and validated. The validation error is used to compare one hyperparameter set with another, but cannot be used as the estimate of model generalisation ability as it played a role in the model building. Cross validation can be used at this stage too (though this can become computationally expensive as part of a grid search) to achieve a good estimate of the differences among hyperparameter sets. Using cross validation at both levels in this way is called nested cross validation.

2.1.6 *Deep Neural Networks*

As discussed in section 2.1.4, it can be beneficial to use several hidden layers in a neural network rather than a single large one. This has led to the study of *deep networks* [84] with many hidden layers. Deep networks have been used with great success in fields such as image and sound recognition where domain knowledge has helped to shape the way in which the layers are organised. A common approach to image recognition is to use *convolutional layers*, which consist of hidden units connected to a small region of the previous layer (known as a receptive field). The input, for example, is split into many overlapping receptive fields, the nodes of each are connected to a single node in the next layer. Each hidden node shares the same weight values as all the others, so the layer represents a moving feature detector. Several such layers will be connected to the layer that precedes them, implementing a bank of different filters. Convolution

layers are often followed by pooling layers that summarise (for example calculating maximum) across a region in the feature map (i.e. node outputs) of the previous layer.

Deep networks also contain one or more layers that look the same as those from an MLP, fully connected to the nodes above and below. As with MLPs, training in deep networks descends an error gradient.

STRENGTHS AND LIMITATIONS The exact behaviour of the filters is learned from data, but there is more room for human design in deep networks than there is in MLPs. This fact also allows deep networks to escape some of the black box limitation as methods for understanding and visualising the function of such networks have been proposed [151]. Deep networks have been very successful recently in fields of computer vision [79] and speech recognition [36]. These tasks are quite removed from the regression tasks described here, though there is no reason why a deep network couldn't be used as a regression model. Features such as convolving on limited receptive fields are obviously specifically designed to solve signal classification type problems where they can be used to overcome location invariance. Deep neural networks also require care to avoid over fitting during learning. Bengio [13] offers a practical guide to training deep neural networks, which addresses the risk of over fitting.

2.1.7 *Regression Trees*

All of the methods discussed so far have had the global property that a parameter interacts with a variable in exactly the same way regardless of the value of all other variables. Interactions are handled with the introduction of feature detectors such as hidden units, or (in the case of linear regression) ignored. In contrast, regression trees [19] partition the data so that the values of regression parameters are different from one partition to the next. The local property of the regression model for each partition means that interactions among inputs have an effect on the output defined by the partition into which they fall.

REPRESENTATION Partitions of varying sizes are defined in a hierarchy, forming a tree consisting of branch nodes, which identify a variable and possess a branch for each value it can take, and leaf nodes, which contain a regression model mapping the variables in the path leading to that node to the output.

LEARNING ALGORITHM Regression trees are often built using greedy algorithms that pick a single variable on which to partition the data at each branch.

STRENGTHS AND LIMITATIONS Regression trees are easy to interpret for a human reader. The upper most nodes tell you which variables are most important and the regression models at the leaf nodes can be simpler than a global model as fewer variables may be needed.

2.1.8 *Basis Functions*

If every function in a given space can be uniquely constructed as a linear combination of a set of orthogonal functions, that set of functions is known as a basis and each of its members is a basis function. One such basis in the space of $f : \{-1, 1\}^n \rightarrow \mathbb{R}$ is the Walsh basis [142], [12], which is of particular relevance to this work because it explicitly encodes the way interaction among inputs affect the output of a function.

REPRESENTATION The Walsh basis for a function of n variables consists of 2^n Walsh functions and each one relates to a subset of input variables. Each Walsh function, ψ_j maps a vector $X \in \{-1, 1\}^n$ to a value in $\{-1, 1\}$. Associated with each ψ_j is a coefficient, $\omega_j \in \mathbb{R}$. Any function, $f : \{-1, 1\}^n \rightarrow \mathbb{R}$ can be rewritten uniquely as a weighted sum of Walsh functions where the coefficients, ω_j are the weights using the inner product

$$f(X) = \sum_{j=0}^{2^n-1} \omega_j \psi_j(X) \tag{2.26}$$

where

$$\psi_j(X) = \oplus(X \wedge j_{bin}) \tag{2.27}$$

where $\oplus(X)$ is a parity count function that returns 1 if the number of values set to 1 in X is even and -1 otherwise and j_{bin} is the binary vector representation of the index j. The $X \wedge j_{bin}$ uses a binary vector representation of j to select the subset of variables in X operated on by $\psi_j(X)$.

LEARNING ALGORITHM The Walsh coefficients, ω_j are calculated by summing the product $f(X)\psi_j(X)$ over all possible instantiations of X.

$$\omega_j = \frac{1}{2^n} \sum_{X \in \{1,-1\}^n} f(X)\psi_j(X) \tag{2.28}$$

STRENGTHS AND LIMITATIONS Any function in $f : \{-1, 1\}^n \rightarrow \mathbb{R}$ can be represented by a Walsh decomposition, so there is no limit on the complexity of functions that can encoded in this way. The decomposition also has the desirable feature of explicitly revealing information about the complexity of the encoded function. Each ω_j is responsible for one subset of input

variables and defines how their interaction (in terms of parity) affects the function output. Any subset of variables that do not interact will have a coefficient of zero so both the number of non-zero parameters and the size of the subsets they represent are made clear.

The disadvantage of any basis method including Walsh is that an exhaustive sample of the (input,output) space is required to calculate the coefficients. Similarly, with 2^n coefficients to calculate, only models with a small number of inputs (around 20 or fewer) may be considered before the computational resources required grow prohibitive. Additionally, the decomposition assumes that the values of $f(X)$ are noise free and known.

By considering only the Walsh coefficients that relate to the subsets of input variables of size one, it can be seen that equation 2.26 takes the same form as the linear regression sum in equation 2.4. If the underlying function is linear and the samples noise free, then the regression coefficients are equal to those in the Walsh sum.

2.2 Meta-Heuristic Optimisation

Regression is concerned with learning a function that maps inputs to an output. A related challenge is to find values across the inputs that lead to a desired output. In the field of heuristic optimisation, problems are generally formulated as a fitness function, $f(X)$ that can be evaluated for any instantiation of X. The goal is to discover a particular instantiation of X, let's call it x^*, that causes the output of the function to meet (or be as close as possible to meeting) a stated goal. The goal might be to cause $f(X)$ to output a particular value, or to treat $f(X)$ as a score to be maximised or a cost to be minimised. Heuristics are adopted when optimising $f(X)$ cannot be solved analytically. When the function to be optimised is a system in the real world (a machine, a schedule, a product design, a timetable, etc.) the function may be costly to evaluate and so a computer simulation of the system is used instead. In some cases this simulation is programmed from known rules that govern the system's behaviour and in other cases it can be learned from data reflecting input,output pairs observed from the system.

Depending on how these simulation models are represented, a distinction is made between black box models, in which nothing about the representation of the rules or the function can be used to guide the search and grey box methods, where there are clues in the model that can guide the search. Parametrised models such as multilayer perceptrons and linear models like those proposed in this thesis are grey box models because they provide information such as linkage among inputs and derivatives at all points in the search space.

Optimisation can be framed as a problem of multiple constraint satisfaction. Some constraints are defined in a 'strong' way, meaning they *must* be satisfied. Integer linear programming [116], for example, defines a problem in terms of an expression to be maximised subject to a set of linear constraints. Alternatively, constraints may be defined in a 'weak' sense, usually

with an associated strength, which incur a cost when violated. In the weak case, optimisation is equivalent to minimising the cost of violated constraints. Some grey box models, such as those proposed in this thesis, make the role and the strength of the constraints explicit in their representational structure.

A parametrised model of the fitness function can be useful when the real world system can be sampled but the rules governing its behaviour are unknown. This can be true simply for practical reasons as it is easier to automate the search process in software if the function to be evaluated can be run in code, rather than tested in the real world. It can also be desirable in cases where the function to be optimised can be modelled in fewer samples than it takes to optimise it without a model. Additionally, even if a black box model is available in software, a grey box equivalent may be able to guide the search more efficiently.

The mechanism for choosing a new solution based on those previously tried is known as a *heuristic*, hence the title *heuristic optimisation*. Heuristics are generally applied to a particular search problem, for example the Christofides heuristic [31] for the travelling salesman problem. When the nature of the function being searched is unknown, a heuristic that can work despite such ignorance is required. Such algorithms are known as meta-heuristics.

2.2.1 *Local Search*

Local search (LS) [91] methods form a broad and well studied set of heuristics for searching fitness functions. Local search begins with a single candidate solution and makes progressive improvements by applying a series of local modifications. Each candidate solution has a neighbourhood of states that can be reached in a single local modification step. When no member of the current candidate solution's neighbourhood offers an improvement, the algorithm is said to have reached a local optimum (which may also be a global optimum). Consider the two main components of any local search:

1. **Neighbourhood**: The scope of the local modifications defines the neighbourhood of each candidate solution. Smaller neighbourhoods take fewer evaluations to search but may lead more readily to local optima. Larger neighbourhoods lead to fewer local optima at the cost of requiring more evaluations to be searched exhaustively.

2. **Acceptance Criteria**: The local optimum escape and avoidance mechanism defines what steps the algorithm takes to move away from local optima and avoid returning to them.

The neighbourhoods in a binary search problem can be defined in terms of Hamming distance. Let $N_k(x)$ be the set of neighbouring points to x with a Hamming distance of exactly k. There are $\binom{n}{k} - 1$ points in $N_k(x)$ and the total number of neighbours at all distances from 1 to k is

$$\sum_{j=1}^{k} \binom{n}{j} - 1 \tag{2.29}$$

Local search can proceed by taking the first improving step it finds in the current neighbourhood (first improvement) or by assessing every available neighbour and moving to the one that provides the largest improvement (best improvement). If no improving step is possible, it must make a local optimum escaping step.

2.2.1.1 *Random Restart Hill Climb*

The simplest form of LS involves a local modification neighbourhood containing all the points at a Hamming distance of one from the current point and a mechanism for escaping local optima that involves starting again from a new random point. This is known as random restart hill climb (RRHC) [91]. When searching fitness functions in which acceptable solutions can be reached from only a small number of starting points, RRHC will start in one of those points with very low probability and therefore will most often climb to an unacceptable local minimum. The region from which a hill climb will reach an acceptable local optimum can be enlarged by increasing the Hamming distance that defines a neighbourhood, but at the cost of increasing the number of local steps that are considered at each move.

2.2.1.2 *Iterated Local Search*

Random restarts can be very inefficient as they may lead to the same local optima being reached repeatedly. Additionally, increasing the Hamming distance covered by neighbourhoods can introduce inefficiencies as the size of the search space grows quickly, as described by equation 2.29.

Both of these problems are addressed by iterated local search (ILS) [90], which performs a local search to a local optimum and then makes a larger step in an attempt to escape the local optima but not discard the gains made by the hill climbing steps made so far. The larger step (known as the perturbation) can be made by searching for an improvement in a larger neighbourhood.

By only widening the scope of the search neighbourhood when the current scope is trapped at a local optimum, ILS avoids random restarts and keeps the size of the neighbourhood small during most search steps.

2.2.1.3 *Variable Neighbourhood Search*

A further extension to ILS involves allowing the size of the neighbourhood to grow and shrink dynamically during the search. Variable neighbourhood search (VNS) [98] defines a set of neighbourhoods of increasing size. Local search is performed at the current scope and when that reaches a local optimum, the scope is widened until an improving step is found, at which point the scope returns to the smallest neighbourhood and the process continues. Larger neighbourhoods are only explored when smaller ones are exhausted.

For binary search problems, one approach is to define each neighbourhood by Hamming distance, starting at 1 and increasing the Hamming distance until an improving move is found.

2.2.1.4 *Tabu Search*

One risk associated with performing local search, which ever method is chosen for escaping local optima, is that the same routes will be taken repeatedly, leading back to the same local optima again. The methods described above aim to escape local optima effectively, but Tabu search [46] additionally aims to avoid returning to them. This is done by maintaining lists of previously visited solutions that are to be avoided, lists of promising areas to be explored further and rules that force diversification into unsearched areas. The list of solutions to avoid effectively re-defines the search neighbourhood of each point to exclude those in the list.

2.2.1.5 *Simulated Annealing*

Another way to allow an algorithm to escape local optima is to take unimproving steps with a certain probability. Simulated annealing (SA) [112] is an algorithm for avoiding local minima in this way. Rather than improving with every step, SA takes a step with a probability that is proportional to the change that would result. A temperature term controls the mapping between the size of the improvement a step would yield (which might be negative) and the probability of the step being accepted. In many implementations, the probability of accepting an improving step is always 1 and only steps that lower the score are taken with a lower probability. The probability of any particular step being accepted depends on the change it makes to the score, the current temperature and the order in which neighbours are considered.

The temperature starts high and gradually decreases according to a cooling schedule. High temperatures lead to higher probabilities of moves that make no improvement being accepted, compared to lower temperatures. At temperature zero, the process becomes a hill climb.

The neighbourhood considered by SA is generally larger than that considered by other local search algorithms as it has to allow larger jumps in the input space. Li and Ma [88] considered three approaches to defining the neighbourhood for binary problems during SA. The first is an enumeration over all the variables, the second allowed the entire search space to be considered by picking a new candidate point uniformly at random and the third is the same as the second

with a local search from the newly chosen point. The third method was found to be most effective on the test problems described by Li and Ma. In this approach, when a new step is accepted, the algorithm performs a local search until no improvement can be made, and then considers new points chosen across the whole search space with uniformly random probability. New steps are accepted with a probability determined by the current annealing schedule and the change in score that moving to the new point would produce. This approach is a hybrid of ILS in which the perturbation neighbourhood is determined stochastically by the temperature and change in output score.

Another method for choosing the neighbourhood in a simulated annealing search over binary variables involves selecting variables that persistently take the same value (0 or 1) and fixing them in future samples [25].

Simulated annealing is a Markov chain Monte Carlo method inspired by the Metropolis-Hastings method [28] for sampling from an arbitrary distribution. Metropolis-Hastings is designed to take any function, $f(X)$ and produce a Markov sequence of samples that are drawn from a distribution, $p(X)$ that is proportional to $f(X)$. When attempting to maximise $f(X)$, the ability to draw new samples with a probability that is proportional to the output of that function is what motivates the simulated annealing approach. In simulated annealing, the temperature parameter changes the shape of the distribution from which samples are drawn, starting with a distribution that is closer to uniform and moving towards a distribution that is proportional to $f(X)$.

The samples produced by simulated annealing are not independent and the algorithm can become trapped around one mode in a multi modal distribution. The stationary distribution of the Markov chain equals the target probability distribution in the long run, but correlations from one sample to the next mean that bias is introduced in the short run. Additionally, a number of steps (known as the burn in period) are required before the stationary distribution settles to the target distribution. The first samples are discarded (the number varies, but can be over 1000). These facts become important when using simulated annealing to solve problems where the fitness function is costly to evaluate or where the measure of success for an algorithm involves counting the number of fitness function evaluations made.

To avoid the problems associated with correlated samples, methods such as importance sampling, which can generate independent samples, have been proposed [99]. In high dimensional multivariate distributions, choosing each new point across all the dimensions can be difficult so Gibbs sampling may be used instead. This involves choosing a new value for one variable at a time, with each sample being the result of a single change. This is particularly suitable when each variable is conditioned on a small number of other variables and the distribution being sampled is represented in a way that makes that conditional neighbourhood explicit. The next section considers the same motivation, that of generating samples with a

probability that is proportional to their fitness, from a different perspective where models of the fitness function are built and sampled.

2.2.2 *Estimation of Distribution Algorithms*

Simulated annealing attempts to generate samples from a distribution, $p(X)$ where $p(X)$ is proportional to the function being searched, $f(X)$ based on a chain of evaluations of $f(X)$. Rather than discarding each evaluation after it has been used, it may be more efficient to use the samples from $f(X)$ to build a probability distribution model and then sample from that. In cases where evaluating $f(X)$ is expensive and modelling the distribution can be done in fewer samples than the number required to perform simulated annealing, this approach can offer an efficiency gain. Estimation of Distribution Algorithms (EDAs) take this approach.

Rather than model the fitness function exactly, EDAs generally take an iterative approach and attempt to model a distribution of promising candidate solutions only. In general, an EDA proceeds as follows. A set of samples of X, $f(X)$ are taken and the fitter solutions among them are selected. A model is built that represents the distribution of values among the selected solutions and the process repeats, with the new set of samples being drawn from the latest model.

The difficulty in this approach is the need to choose the correct model for the distribution. This is a question of choosing the correct model bias and there are generally trade-offs among the complexity of the model, the number of samples required to fit its parameters, and the number of modes (local optima) the distribution can represent. Many EDAs employ graphical models to represent the joint distribution and so must represent interactions among inputs explicitly.

The extended cGA (ECGA) [51] models higher order interactions by searching for a marginal probability model (MPM) which models the joint distribution of several non-overlapping variable partitions. The partition set is chosen using a greedy algorithm, which begins with a model in which all partitions are of size 1 and then performs a steepest descent search by merging the single pair of partitions which decreases minimum description length the most at each step. An algorithm called mutual information maximising input clustering (MIMIC) [16] represents model structure as a chain of conditional probabilities. In [9] the model is represented as a dependence tree, as it is in the COMIT algorithm [8], which is an extension of MIMIC that uses a tree structure with the addition of a hill climbing phase to optimise the individual solutions it creates.

The problem with such algorithms is that they impose a constraint on the structure that may not be suitable to the function being modelled. In other words, they all (somewhat arbitrarily) introduce model bias. One notable exception to this is DEUM [117], which builds a distribution in the form of a Markov Random Field (MRF). DEUM is attractive because it allows model

bias to be learned from the data. The MRF can be sampled using Gibbs sampling to generate candidate solutions with a probability that is proportional to their fitness scores. Methods for discovering structure in Markov random fields are discussed in section 2.4.

Similarly, hierarchical Bayesian optimisation (hBOA) [104] builds a Bayesian network and uses it to generate new candidates in an evolutionary search. The hierarchical aspect is that hBOA replaces the conditional probability table that is usually used in each node of a Bayesian belief network with a local decision graph, reducing the complexity at the node and consequently, the size of the data set required to build the network.

The key ingredient of EDAs is the use of selection to restrict the space over which the distribution is accurately modelled. Depending on the nature of the function being searched, there are advantages and disadvantages of making use of selection. An potential advantage is that the models may be simpler than they would need to be to model the entire input space of the function accurately. This may reduce the number of fitness evaluations needed to build a model. A potential disadvantage is that many of the fitness function evaluations are discarded as only the fittest solutions are chosen for modelling. This, coupled with the need to iterate the algorithm over several generations can increase the number of fitness evaluations needed. The other side of this trade-off would involve discarding the use of selection and using every single fitness evaluation made. This could also remove the need to iterate over more than one generation. It would, however, require a more accurate model of the fitness function. These approaches lead us to the use of fitness function models.

2.2.3 *Fitness Function Models*

Fitness function models (FFMs), (sometimes called emulators or surrogate fitness functions), aim to reproduce the fitness function in a form that has benefits over its existing form. It may be that the model is faster to evaluate or easier to differentiate or reveals insights into the structure of the real function. FFMs are also very useful when a fitness function may not be programmed and the only available guide to its behaviour is a data set of (input, output) samples. Some methods involve building a model from a single sample of data and others involve iterations that alternate between sampling the fitness function model and the real fitness function.

A number of different machine learning methods have been applied to fitness function modelling, including multilayer perceptrons [64] and Kriging [144]. Jin provides a good review of neural networks as FFMs [70] and describes a framework for using approximations to fitness functions [147]. Gaussian processes have become popular methods for modelling fitness functions [110], but they are designed for use on continuous variables rather than the binary inputs under consideration in this work.

FFMs can be used in a variety of different ways. A sufficiently good model of the data can be searched without reference to the original function. A simpler model can be used as a filter

to rule out really low scoring candidates as a way of reducing evaluations by the real fitness function or to suggest regions that might profitably be explored. The models themselves must be searched, and so a search heuristic is still needed—the model is not a heuristic in its own right. Consequently, it is highly desirable for the representation used for a FFM to transparently reflect something of the structure of the function being searched. For example, it may be useful if parameters in the model represent identifiable aspects of the function's behaviour. The model can then provide guidance to the heuristic algorithm that is unavailable from a black box model. A key feature of the methods proposed in this thesis is their ability to provide such guidance. The right kind of FFM can be considered as a method for turning black box problems into so called grey box problems, where the structure of the model representation guides the search process.

2.2.3.1 *Heuristic Evaluation*

The quality of the performance of an optimisation heuristic should be measured over several attempts at solving a problem and may be measured in terms of the average number of fitness function evaluations or time taken to find a solution or a measure of solution quality such as its best score over all the runs or the number of times the global optimum was found.

2.3 Dynamic Systems

The preceding sections have described feed forward systems where the inputs affect only the outputs. It is also possible to treat all variables equally as inputs and outputs, using only the vector X such that any $X_i \in X$ has a value that is dependent on zero or more of the other variables. The behaviour of these dynamic systems as values are updated is more complex than that of a feed forward system. Such systems are closely related to feed forward systems in the sense that the connections among the variables can be viewed as constraints that determine what values those variables should take, given the values of their neighbours. The extent to which any given input state, x is consistent with those constraints is often referred to as the state's energy and finding the state with the minimum energy (i.e. the greatest degree of agreement among values and their constraints) is an example of heuristic optimisation.

2.3.1 *Graphical Models*

A graphical model of a function represents the structure of the function as a graph and a set of parameters. The graph, $G = (N, E)$ consists of nodes, N and edges, E. The nodes represent variables and the edges represent dependencies between those variables. The parameters can be directly associated with the edges so that each edge has a corresponding parameter, or the

parameters can be associated with other features of the graph such as the neighbours of each node.

2.3.1.1 *Hopfield Networks*

A Hopfield network (HN) [66] is a type of neural network with input variables represented by nodes in the network but no output variables. Its graphical representation is formed such that the set of nodes, N represent the set of input variables in X and the edge set E contains weighted connections between pairs of nodes. The weighted connection between X_i and X_j is denoted $W_{i,j}$. Weights are symmetrical so $W_{i,j} = W_{j,i}\ \forall i, j$ and there are no self connections so $W_{i,i} = 0\ \forall i$. Each node has an associated value, known as its output and the values across all of the nodes, x represents the network's current state. The output of a node is calculated from the outputs of the other nodes by first summing the product of the incoming weights and the output of the nodes they connect. This calculates the node's activation, which is then passed through a threshold function so that $x_i \in \{-1, 1\}$. The activation is

$$a_i = \sum_{j=0}^{n-1} w_{ji} X_j \tag{2.30}$$

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

$$X_i = \begin{cases} 1 & \text{if } a_i > \theta \\ -1 & \text{otherwise} \end{cases} \tag{2.31}$$

The dynamics of the network are the result of repeatedly applying equations 2.30 and 2.31 for a selected node in the network. Nodes are selected uniform randomly without replacement so each node is updated once in a single pass, at the conclusion of which, the pool of available neurons becomes the full set once more.

Each network state has an associated energy function, given in equation 2.32. The weights between nodes can be viewed as constraints where the sign of the weight dictates whether the constraint is that the connected nodes should be equal (positive weight) or different (negative) and the magnitude of the weight measures the strength of the constraint. In this light, the energy of a given network state reflects the degree to which the weighted constraints are satisfied. The global minimum of the energy function is produced when the network is in the state that best satisfies the constraints defined by the weights. Local minima in the energy function are states from which applying equations 2.30 and 2.31 cannot cause a change in any neuron's value.

$$U(X) = -\sum_{i,j} W_{ij} X_i X_j \tag{2.32}$$

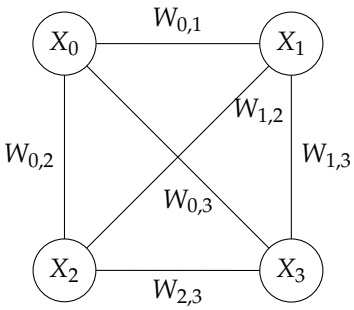

Figure 2.2: A four neuron HN with units X_i and weights $W_{i,j}$.

Hopfield networks have most commonly been used as content addressable, or error correcting memories. A pattern, x across the inputs is learned by updating the weights using the Hebbian rule as follows

$$W_{ij} \leftarrow W_{ij} + x_i x_j \quad \forall i \neq j \tag{2.33}$$

The symmetrical weight connections and zero self-connections mean that the energy function of equation 2.32 is a Lyapunov function, meaning that the dynamic of equations 2.30 and 2.31 always lead to a stable state, known as an attractor. The purpose of the Hebbian learning rule of equation 2.33 is to create attractor states at desired input points, known as memories. An attractor state is a local minimum in the energy function. A Hopfield network has only pairwise connections (higher order networks are considered next) so has a model bias that limits the set of functions that can be modelled. This restricts the number of turning points the modelled function can represent, which limits the number of memories the network can store. This limit is often referred to as the network's capacity.

The addition of a memory state using equation 2.33 has the effect of adding a local minimum to the energy function at $X = x$ [95]. The set of energy functions that a Hopfield network can represent is limited, so there is a limit on the number of local minima it can store, known as its capacity. For uniform randomly generated patterns learned with the Hebbian rule, McEliece [95] stated that the capacity of a fully connected n unit HN trained with the Hebbian update rule is $n/(4 \ln n)$ and that capacity improves to $n/(2 \ln n)$ if a small amount of degradation in the stored patterns is tolerated.

2.3.1.2 *Higher Order Hopfield Networks*

Neurons in a standard HN are connected in pairs, but it is also possible to connect groups of 3 or more neurons with a single weight. Adding higher order weights to the network reduces model bias and expands the set of functions that can be modelled. Kubota [81] states that the capacity of fully connected order m associative memories is $O(n^m / \ln n)$. Venkatesh and Baldi [140] report that regardless of the learning algorithm, the capacity of a high order network is of

the order of 1 bit per weight. For the outer product learning rule, this is of the order $n^m / \ln n$, as stated by Kubota [81]. Venkatesh and Baldi also give lower bounds on the capacity of high order Hopfield networks [141] with weights set using the outer product rule with a zero diagonal as

$$\underline{C_n} \approx \frac{n^d}{d!2(2d+1)\ln n} \tag{2.34}$$

and the weak lower capacity, $\underline{C_n^w}$ (allowing a small number of erroneous recalls) is

$$\underline{C_n^w} \approx \frac{n^d}{(2d+1)!\ln n} \tag{2.35}$$

These capacity calculations are based on memories sampled from a series of symmetric Bernoulli trials.

Shen et al. [118] showed that high order HNs converge to equilibrium points in the same way that second order HNs do. Samad and Harper [114] used high order HNs to solve the graph partitioning problem, in work that combines high order networks and their application to optimisation. As with the other work reviewed here, this work used a hand coded network, which was designed specifically for the task at hand.

2.3.1.3 *Boltzmann Machines*

Replacing the threshold neurons in a HN with stochastic neurons and introducing hidden units produces a network known as a Boltzmann machine [1]. The hidden units are fully connected to each other and also fully connected to the input units, so their structural role is identical to that of the units in a HN. The role they play during learning and running, however, is different. Neuron updates are stochastic, based on the conditional probability of the target neuron taking a value of 1 given the current values across the rest of the neurons.

Let $P(X_i = 1|X_{\setminus i})$ denote the conditional probability of neuron X_i taking the value of 1, given the current values across the rest of the nodes, $X_{\setminus i}$. This is defined as

$$P(X_i = 1|X_{\setminus i}) = \frac{P(X_i = 1, X_{\setminus i})}{P(X_i = 1, X_{\setminus i}) + P(X_i = 0, X_{\setminus i})} \tag{2.36}$$

which can be written as

$$P(X_i = 1|X_{\setminus i}) = \frac{e^{\sum_{j \neq i} w_{ij} X_j}}{1 + e^{\sum_{j \neq i} w_{ij} X_j}} \tag{2.37}$$

and simplifies to

$$\frac{1}{1 + e^{\sum_{j \neq i} w_{ij} X_j}} = \frac{1}{1 + e^{-U}} \tag{2.38}$$

which is the logistic function used in logistic regression and multi layer perceptrons. Boltzmann machines do not settle into a single attractor state like HNs, but settle instead into a distribution of states, moving from one to the next by changing neuron states by equation 2.38. This is derived from the Boltzmann distribution that gives the network its name and defines the probability of each possible state the network might take as a function of its energy:

$$P(X) = \frac{1}{Z} e^{-U(X)} \tag{2.39}$$

where Z is the sum over all possible patterns:

$$Z = \sum_{X} e^{-U(X)} \tag{2.40}$$

and the energy, $U(X)$ is calculated in the same way as for the Hopfield network

$$U(X) = -\sum_{i,j} W_{ij} X_i X_j \tag{2.41}$$

2.3.1.4 *Strengths and Weaknesses*

The hidden units extend the power of the Boltzmann machine, allowing it to model a far greater range of energy functions but, like the hidden units in an MLP, they obscure the true structure of the function. The Boltzmann machine learning algorithm involves finding the set of weights that cause the equilibrium distribution of the network to represent as closely as possible the true distribution of the data. To learn a sufficient number of samples and generated patterns, the algorithm has to run the network to equilibrium. This is a time consuming process and has stifled the application of Boltzmann machines to real problems. One solution to this problem is the restricted Boltzmann machine, which is described next.

2.3.1.5 *Restricted Boltzmann Machines*

In restricted Boltzmann machines (RBMs) the visible neurons are separate from the hidden units, forming a bipartite graph. Figure 2.3 shows a RBM with four visible neurons and four hidden. The visible neurons are v_i, the hidden are h_j, the connections between the two layers form the weights w_{ij}. Units also have incoming bias weights, those on the visible neurons are a_i and those on the hidden are b_i.

The energy of the network is calculated as

$$U(v,h) = -\sum_{i} a_i v_i - \sum_{j} b_j h_j - \sum_{i} \sum_{j} h_j w_{ij} v_j \tag{2.42}$$

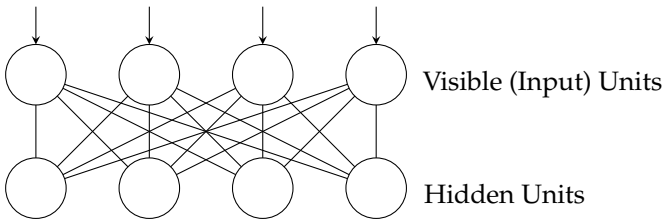

Figure 2.3: A restricted Boltzmann machine with four visible and four hidden units.

As with the Boltzmann machine, the probability of a pattern over the visible neurons is calculated by marginalising over the hidden units:

$$P(v) = \sum_h \frac{1}{Z} e^{-U(v,h)} \tag{2.43}$$

The weights are symmetrical and there are no connections between pairs of visible units or pairs of hidden ones. These facts allow the probability of a visible pattern to be calculated conditional on the hidden units alone, and the probability of a pattern across the hidden units to be calculated conditional on the value of the visible units:

$$P(v|h) = \prod_i P(v_i|h) \tag{2.44}$$

$$P(h|v) = \prod_j P(h_j|v) \tag{2.45}$$

where in each case the individual neuron probabilities are calculated using the sigmoid activation function:

$$P(v_i = 1|h) = \frac{1}{1 + e^{-a_i}} \tag{2.46}$$

where

$$a_i = b_i + \sum_j w_{ij} h_j \tag{2.47}$$

A RBM can be trained using the contrastive divergence algorithm (CD) [57], which uses a maximum likelihood approach that attempts to maximise the product of the probabilities over the visible units. Hinton provides a useful guide to training RBMs [58]. Hinton has also shown how RBMs can be layered to produce deep belief networks using the same learning rule [59]. Though powerful, these networks use hidden units that obscure the relationships between variables in a manner similar to that of the MLP.

2.3.1.6 *Markov Random Fields*

Hopfield networks can be sparsely connected but the canonical model has weights connecting every neuron pair. Weights are attached to the connections so the shape of the energy function that can be modelled is limited (which is why such networks have a limited capacity as content addressable memories). Boltzmann machines attempt to solve this limitation by introducing hidden units. An alternative graph modelling approach is to let the edges of the graph dictate dependence between variables but move the function parameters away from the edges. A Markov random field (MRF) is used to model the joint probability distribution of a set of n variables, X based on dependencies between pairs of variables. A MRF is defined by the graph, G that describes its connections and a parameter set, θ that defines its energy function. The description provided here applies to MRFs where each $X_i \in \{-1, 1\}$, but the theory extends to larger discrete sets of values and continuous variables.

The graph, $G = (X, E)$ representing a MRF consists of a vector of variables, X and a set of edges, E joining pairs of variables. The subset of variables that are all connected to X_i are known as the neighbours of X_i, denoted N_i. Note that $X_i \notin N_i$. The Markov property of a MRF means that any variable is independent of any other variable that is not in its neighbourhood. Any subset of X in which every variable is connected to every other is called a clique. Larger cliques must contain smaller sub cliques, and any clique that is not a sub clique is called a maximal clique.

Let $C \in G$ iterate over the maximal cliques in graph G. For each clique, C there is an associated clique potential function $V_C(X)$ which operates on the members of X that belong in clique C. The energy $U(x)$ associated with a network state, x is the sum of the clique potentials

$$U(x) = \sum_{C \in G} V_C(x) \tag{2.48}$$

According to the Hammersley-Clifford theorem [50], a MRF can be represented as a Gibbs distribution, which takes the form

$$P(X) = \frac{1}{Z} e^{-\frac{1}{T}U(X)} \tag{2.49}$$

where Z is the sum of equation 2.49 over all possible instantiations of X and T is a temperature control that determines the steepness of the function.

When building a MRF from data it is necessary to infer the correct structure for the graph, the structure of the clique potential functions within it, and the parameters for those functions. The question of structure discovery is considered next.

2.4 Structure Discovery

The previous sections described how a linear regression model is simple to train but can lack the required complexity to model many real world relationships. Multi layer perceptrons were presented as a solution to this problem, but with the disadvantages associated with the fact that the MLP's search for and representation of those complex relationships are opaque. Walsh functions were described as a method for explicitly representing arbitrary complex relationships but with the disadvantages of requiring a full and noise free sample of the data and exponentially many parameters.

By choosing a function representation that makes the relationships among variables explicit, the advantage of human interpretability is gained at the expense of needing to discover the correct structure. Finding the right structure is generally an NP problem. This section reviews a number of approaches to discovering function structure explicitly. Much of the recent work on structure discovery has been carried out in the field of meta-heuristic search, described in section 2.2.

2.4.1 *Linkage and Building Blocks*

It has long been understood that the effectiveness of evolutionary optimisation meta-heuristics is dependent to a large degree on their ability to preserve important building blocks from one generation to the next. A building block [47], [62] is a subset of variables that interact to improve the fitness of a solution. Each variable may seem to have little to contribute on its own, but may be crucial in producing a solution with high fitness when combined with others in its block. This effect is known as epistasis [37]. Genetic algorithms can easily break good building blocks apart during recombination, which has led to a number of attempts at controlling the crossover and ordering of variables in a GA [72].

Let us consider two approaches to coping with Epistasis: linkage identification methods (sometimes called perturbation methods) and EDAs. Linkage identification methods, [73], [54] look for relationships by comparing the effect on fitness of flipping each variable from a chosen pair in isolation to the effect of flipping both together (sometimes known as probing). In theory, the approach could be extended to higher order linkages, but the number of fitness function evaluations required would soon become impractical. The other problem with such approaches is that the effect of flipping the pair of variables might itself be dependent on the values of other connected variables, so the test is not conclusive. Streeter [123] presents an algorithm that makes use of a binary search to discover linkages and proves that it is capable of discovering the linkage structure of any additively separable function of n inputs in $O(2^k n \ln(n))$ fitness function evaluations, where k is the size of the separate sub functions.

Coffin and Smith [32] point out that most EDA searches are greedy and start from a search for pairwise interactions that can fail to find higher order interactions that are not signalled by similar ones at lower orders. Consider the Walsh basis discussed in section 2.1.8. Of the 2^n basis functions, only $\binom{n}{2}$ are pairwise, leaving the majority undiscoverable by many current algorithms in the absence of signposts from the second order coefficients. They suggest that researchers 'bite the bullet' and search for higher order linkages or employ a hybrid method involving both an EDA and linkage detection such as D^5 [138].

DEUM [117] follows a similar greedy path from second order to higher order weights, but does so in a single step. It forms a graph that connects pairs of dependent variables, up to a maximum number of neighbours per variable. Then it finds all of the maximal cliques in the resulting graph and a choice is made as to which of those cliques and their subcliques to include in the model. If all of the subcliques are included, the number of parameters in the model grows exponentially with the size of each maximal clique. Two pitfalls for the algorithm are that it relies on pairwise interactions as the starting point for finding those of higher order and that the number of candidate cliques may not be significantly reduced if the cliques are large.

Further work [94] proposed Sparsified DEUM (sDEUM) which used L1 regularisation to remove weights from the maximal cliques. This approach requires a data set at least equal in size to the number of weights produced by the full connection of the maximal cliques.

2.4.2 *Bayesian Belief Networks*

Bayesian belief networks (BBNs) model joint probability distributions as a network of conditional probabilities. Their representation is a graph where nodes correspond to variables and edges indicate the presence of a conditional relationship between two variables. Conditional probability is not symmetrical ($P(X|Y) \neq P(Y|X)$) so the graphs are directed and no variable can be conditional on its own value, even as the result of a cycle, so Bayesian networks are directed acyclic graphs. Bayesian networks are of interest in this context because their connectivity structure must be discovered from data.

Discovering the correct structure for a general discrete Bayesian network from data is NP-hard [30] and has received a lot of research attention. There are exact solutions based on dynamic programming [101], for example, but they have complexity of $O(n2^n)$ and are impractical where n is in the mid tens or more. Approximate methods mostly rely on imposing limits or constraints on the structure's complexity to reduce the search space. Early methods such as the K2 algorithm [33] were based on a predefined ordering of the variables and a limit on the number of parents any node could take (essentially a limit on order complexity) and so reduced the search to polynomial time at the expense of the bias introduced by these two restrictions.

The way in which K2 limits the complexity of a network is crude–a simple limit on parent numbers and an ordering of variables. An alternative method is to use minimum description

length (MDL). An MDL approach that still requires the variables to be ordered has been proposed in the K3 algorithm that replaced the K2 measure with one based on MDL [18]. Branch and bound methods have been used to attempt to reduce the search space in conjunction with MDL, [124] and with linear programming [69].

More recently, de Campos and Ji [38] propose a branch and bound method and use structural constraints to reduce the search space. They also point out that a good search method should produce what they call an *anytime* solution, which means that the algorithm iteratively improves the quality of the solution and may be stopped at any time, rather than needing to run to completion before a solution is available.

Researchers have addressed the DAG discovery problem with a range of machine learning and optimisation methods such as a GA to search the space of orderings before applying the K2 algorithm, [82], greedy search which adds, removes or reverses connections at each step, [55], evolutionary programming and MDL, [148] and the HEP algorithm [149], which used statistical independence tests and evolutionary computation.

2.4.3 *Multi Layer Perceptrons*

The standard structure of an MLP contains an input layer, one or more hidden layers and an output layer. Each layer is fully connected to the neurons in the layer above. Approaches to dynamically changing the structure of an MLP during training involve adding or removing weights or neurons and their associated set of weights. Bartlett [10], for example proposed an algorithm that added hidden units each time the training error flattened, and removed units based on an information theoretic measure. He also pointed out that the network weights were often optimised to the structure, and adding new ones didn't allow the network to escape the local optimum it was in. LeCun et al. [85] proposed the Optimal Brain Damage algorithm, which removes weights with low saliency, which is defined based on the second derivative of the cost function. Some algorithms continue to train all of the weights after each iteration of adding or removing weights. Others, such as DMP3 [3] freeze existing weights and only train the newly added ones. Some algorithms add neurons in a restricted structure, for example in the Upstart algorithm [42], the network becomes a tree structure as new neurons are added below existing parent neurons. Although not strictly a structure discovery approach, dropout [121] is a method that drops random neurons during training and then approximates the average output of all the resulting smaller networks at test time. There have also been evolutionary approaches to MLP structure discovery, for example Garcia [45] introduced a new crossover operator to allow a GA to discover MLP structure, solving the permutation problem (being that network structure tells you little about network function). See [150] for a review of evolutionary approaches to neural network learning.

2.4.4 L_1 *Regularisation Methods*

Observing that L_1 regularisation methods such as the lasso combine feature selection and parameter fitting has led researchers to investigate their use for structure discovery in graphical models. Schmidt, for example, [115] builds dense models and then uses L_1 regularisation to induce sparsity. Most of the work proposed in that work limits the dense model to pairwise connections but hierarchical restrictions are also proposed to allow higher order interactions to be considered. In a hierarchical model, if a coefficient β_a is zero, then the coefficient for all higher order interactions, β_b for $a \subseteq b$ are also zero. Non-hierarchical models have the disadvantage of not being invariant under reparameterisation. It is possible in a non hierarchical model in which $\beta_a = 0$ and $\beta_b \neq 0, a \subseteq b$ for an addition or removal of some parameters to cause $\beta_a \neq 0$. The disadvantage of a hierarchical approach is that it can only discover the subset of models that are hierarchical in nature.

Most approaches consider only pairwise interactions, one exception being Dahinden et al., [35] who consider higher order interactions, but only on small numbers of variables. Considering a large model, which is then pruned has the disadvantage of requiring a training data set that is large enough to fit the dense model even when the sparse model could be fitted with a much smaller sample, if only its structure were known.

The lasso regularisation has also been used to discover the zero valued coefficients in the covariance matrix (and, consequently, a sparse dependency graph) by considering the neighbourhood of each variable in turn, adding both nodes and connections as the size of the data set grows to allow them [96]. The ability to match the number of parameters in a model to a level supported by the data is an important consideration in structure discovery.

L_1 regularisation was also used to discover structure in Markov networks by Su-In Lee at al. [87] who employed a greedy method of adding weights and relied on L_1 regularisation forcing some weights to zero when choosing weights to remove. The approach maintained two sets of features: one active, with parameter values set by gradient descent and the other inactive with parameter values fixed at zero. Two greedy feature introduction methods were considered, both of which choose a single feature to add from the complete inactive set. The grafting procedure [106] chooses the parameter with the steepest gradient with respect to the cost function and the gain-based method [39] calculates the gain in log-likelihood of adding each parameter. In each case, once the feature has been selected, the parameters across the new active set are relearned and the process is continued. Su-In Lee at al. point out that considering the effect of every inactive weight in large networks is prohibitively expensive and suggest that a smaller inactive set should be pre-selected, with the remaining weights ignored entirely.

The grafting procedure for feature selection [106], which has been used for Markov network structure discovery [87] combines regularisation with greedy feature introduction. A regularised cost function is defined and gradient descent is used to find the global minimum for an active

feature set. New features are added one at a time, selected by the partial derivative of the cost function with respect to each candidate feature.

The loss function associated with the weight set $\mathbf{W}$ over the training data set of size m is the mean of squared errors

$$L(\mathbf{W}) = \frac{1}{2m}\sum_{j=1}^{m}(f(x_j) - \hat{f}(x_j))^2 \tag{2.50}$$

and the cost function is the loss plus a scaled sum of the absolute size of the weights

$$C(\mathbf{W}) = L(\mathbf{W}) + \lambda \sum_{\omega \in \mathbf{W}} |\omega| \tag{2.51}$$

The derivative of the loss function with respect to weight ω_i is

$$\frac{\partial L}{\partial \omega_i} = \frac{1}{m}\sum_{j=1}^{m}\left(\frac{\partial \hat{f}(x_j)}{\partial \omega_i}\frac{\partial L}{\partial \hat{f}(x_j)}\right) + \lambda \text{sign}(\omega_i) \tag{2.52}$$

and the derivative of the cost function with respect to weight ω_i is

$$\frac{\partial C}{\partial \omega_i} = \frac{\partial L}{\partial \omega_i} + \lambda \text{sign}(\omega_i) \tag{2.53}$$

Every candidate weight will have a value of zero and its impact on the derivative is defined as

$$\text{sign}(\omega_i) = \begin{cases} -1 & \text{if } \frac{\partial L}{\partial \omega_i} > \lambda \\ 1 & \text{if } \frac{\partial L}{\partial \omega_i} < -\lambda \end{cases} \tag{2.54}$$

The grafting approach requires a single pass through the training data for each inactive feature under consideration. Calculating the derivative of the cost function with respect to every candidate weight and effectively discarding all but one of them can be very inefficient.

2.4.5 *Structure Discovery As Variable Selection*

Section 2.1.2 describes a number of variable selection methods. By treating the full set of possible connections in a network as potential variables, it may be possible to apply such methods to structure discovery. As discussed above with reference to the grafting approach, greedy methods suffer from the need to consider every potential variable in isolation before the next one is added. This is not possible for large networks as there are too many potential edges to

even consider each of them in isolation. Never the less, variable selection inspired approaches to structure discovery have been proposed.

As noted in section 2.1.2, this thesis does not take a Bayesian approach, but some Bayesian structure discovery methods are worth noting for comparison. The spike and slab approach has been applied to structure discovery in Markov random fields, for example. Chen and Welling [26] have recently proposed a Bayesian approach to MRF learning with a spike and slab prior. They motivate the choice of spike and slab over a Laplace or Gaussian as these latter priors do not induce sparse structure. They also show that the L_1 models need strong regularisation, which causes unwanted global shrinkage on all the parameters and consequent under-fitting. The spike and slab method is able to learn sparse connectivity models without leading to under fitting on the parameters that are included. The Bayesian approach expresses uncertainty about the inclusion of edges through the posterior distribution, to which a threshold is applied to make the final binary decision of whether or not to include a weight in a model. The experiments reported in [26] only address pairwise connections, which reduces the search space to a manageable size but does not address the challenge of discovering higher order connections.

2.4.6 *Hypernetwork and HyperGraph Structure Discovery*

This section addresses the some of the work by Zhang et al. using hypernetworks and hypergraphs for classification and pattern generation. Kim and Zhang [75] describe a method for pattern classification that uses a hypernetwork representation to implement a function that classifies patterns in $\{0,1\}^n$. Given n input variables, $\{X_1, \ldots, X_n\}$, a graph $G = (N, V)$ is defined with n nodes in N and a population of hyperedges in V, which forms a multiset (an edge can appear more than once). Each hyperedge in V is created from a single entry in the training data. It is made up of a subset of input variables, each with an associated value in $\{0,1\}$ (known as feature sets) and an associated class label, Y. For example, $\{X_3 = 0, X_4 = 1, X_7 = 1, Y = 1\}$ represents one member of the multiset connecting nodes 3,4, and 7, with values instantiated with 0,1, and 1 respectively and the fact that this example led to class 1 in the training example used to build it.

The structure of the hypernetwork is discovered using an evolutionary algorithm whereby each edge has an associated fitness value and those edges with low fitness are replaced with new hyperedges. The two aspects of particular interest from this algorithm are the methods for choosing which candidate hyperedges to add and how to allocate fitness values to those hyperedges (and consequently which hyperedges to remove). New hyperedges are chosen in a two stage process. First, the order of the edge (i.e. how many nodes it connects) is chosen and then the nodes to be connected are added.

The order of a new hyperedge is chosen by sampling from a discrete probability distribution over the available orders (1 to n). This distribution is initially set to be discrete uniform and is then updated to reflect the distribution of orders found to be useful in the classifier. Once the order, c of a new candidate hyperedge has been chosen, a subset of c input indexes are selected from a uniformly random distribution. A single sample data point is then used to instantiate the feature set of the hyperedge and its associated class label.

Hyperedges are selected for removal by first calculating a fitness value for each based on the number of correct classifications it makes across the training data. Hyperedges with high fitness are preferred to those with low fitness and hyperedges of low order are preferred to those of higher order. In this way, the number of parameters and their order can be controlled.

More recently, the same authors [120] built a hypergraph to classify clinical outcomes in cancer cases. The graph structure is discovered using a Bayesian evolutionary approach that maintains a large population of hyperedges. A prior over the hyperedges is chosen based on mutual information between each variable and the class label and also contains pressure to keep the model compact. Edges are sampled based on training data instances, with the number of nodes being connected and the variables they connect being sampled from the evolving distribution. A likelihood measure based empirically on whether the values on the nodes correctly classify the label on the class node they are connected to is used to remove weights.

Hypernetworks have also been used to generate music [74], predict stock market price movements [11], classify protein interactions [17], and even to learning concepts from cartoons [86].

2.4.7 *Structure Learning Summary*

Two main themes emerge from studying existing approaches to dynamically learning the structure of statistical models such as hypernetworks, MRFs, BBNs, and MLPs. One is the need to restrict the search space, either by restricting the complexity of the model or with branch and bound techniques. The other is that evolutionary methods have been popular as structure discovery methods. Most aim to minimise a cost function explicitly (BIC or MDL for example), but some prune parameters based on the statistics of those parameters, something they also have in common with some feature selection algorithms. In fact, structure discovery can be viewed as a form of feature selection in which the full feature set is made up of every possible interaction and many of the possible features can never be evaluated. L_1 methods have been proposed that combine parameter fitting and structure pruning in a single method. The evolutionary approach to hypergraph classifier building is interesting as it explicitly maintains a distribution over hyperedge orders to guide the search.

Although we may draw inspiration from MLP or BBN structure discovery methods, the most useful methods are those designed to build hypernetworks, for example evolving

hypernetworks and MRF discovery methods such as the clique finding algorithm employed by DEUM and L1 methods such as grafting.

These methods share some common features, from which it is possible to sketch a high level description of a single algorithmic approach, shared by them all. Algorithm 1 outlines the steps.

Algorithm 1 General Steps Common to Selected Structure Learning Algorithms

```
Let M be the set of hyperedges in the current model
Let H be the full set of possible hyperedges
Let C be the set of candidate hyperedges under consideration
C ←⊂ H                          ▷ Choose a subset of hyperedges over which to search
repeat
    S ←⊂ C                      ▷ Select one or more hyperedge from the candidate set
    M ← M ∪ S                   ▷ Add the new hyperedges(s) to the model
    Learn the parameter values of M
    Remove a subset of M
    Revise the scope of C
until Stopping criteria met
```

We can now compare each of the selected methods to the general algorithm above. Consider sDEUM first. In the initial step, all pairwise dependencies are modelled, so $\mathbf{C}$ contains all second order edges and no others and $\mathbf{M} = \mathbf{C}$. An initial edge removal step is performed, based on mutual information. A single hyperedge addition step involves revising $\mathbf{C}$ so that it contains all the hyperedges that fully connect the nodes in each maximal clique in the resulting graph and again $\mathbf{M} = \mathbf{C}$. Parameter values are estimated using lasso, which also dictates the hyperedges to be removed. The algorithm does not iterate, so it terminates at this point.

The method based on grafting, [87] lets $\mathbf{C} = \mathbf{H}$ in its basic form but the authors point out that in reality a subset of hyperedges (i.e. $\mathbf{C}$) must be selected. The work does not provide a method for making the selection, however. Each additional weight set, $\mathbf{S}$ is a single hyperedge selected greedily using the partial derivative of the cost function with respect to each weight in $\mathbf{C}$. The L1 regularised cost function is used to learn the parameter values and select those for removal.

The evolving hypernetwork approach [75] limits $\mathbf{C}$ in two ways. A probability distribution over hyperedge orders provides a bias towards some orders over others and the features connected to each new hyperedge are determined by randomly sampling from the training data. Patterns that do not appear in the training data cannot appear in the hyperedge multiset. Hyperedges are removed based on their contribution to making correct classifications and low order hyperedges are preferred over those of higher orders. The distribution that defines the probability of a hyperedge belonging to candidate set $\mathbf{C}$ starts uniform and is updated to reflect the orders from which high fitness hyperedges are found. The framework outlined in algorithm 1 is used in the next chapter as the basis for a new structure discovery algorithm.

2.5 Search in Graphical Models

Some graphical models such as a Hopfield network or a Markov random field represent a function in a way that allows the parameters of the model to be viewed as weighted constraints. Two neurons joined with a positive weight in a Hopfield network will combine to contribute a positive influence on the energy value when their signs agree and a negative influence when their signs differ, for example. Weights can be considered constraints of varying importance and energy minimisation is the process of finding a set of values across the neurons that maximises the difference between the sum of the weights of constraints that are satisfied and the sum of those that are not. This section describes some methods for searching a graphical model for points that minimise an error function (or perform some other type of optimisation).

2.5.1 *Hopfield Networks*

Hopfield networks have been used to solve tasks such as the travelling salesman problem [65], [145] but weights are set by an analysis of the problem rather than by learning. One notable work [24] studied three artificial optimisation problems using high order HNs and presented an analysis of their approach. Their networks were coded by hand, rather than learned from data. They point out that there is no general analytical methodology for choosing the values of the weights in the network and that local minima in the energy function present a problem for an algorithm that searches such a model.

The traditional method for moving a Hopfield network to a minimum in its energy function is to pick nodes at random and set their value according to the weights and values of connected nodes, as described in section 2.3.1.1. Each node's value is either changed (if the connections demand it) or left the same. To ensure that every node is visited, the random order can be subjected to a restriction that no node can be re-visited until all others have been processed. This approach is referred to as next descent search (the first energy reducing step to be found is taken) with simple Tabu (recently flipped nodes cannot be flipped back until all other nodes have been considered).

2.5.2 *Steepest First Search*

An alternative to next first descent is steepest descent search where the node whose change leads to the greatest change in energy is chosen. This can be made very efficient in a graphical model as the change made to energy by any node is fixed with respect to its weights and only changes in sign when the values of nodes to which it is connected change. By keeping track of the effect of a flip on each node, Whitley et al. [143] proposed a method in which the best

next step can be selected in constant time. Although this work is not presented as a method of searching a graphical model (it uses a Walsh decomposition to keep track of connections), the problem being solved is the same. The approach is extended by Chicano et. al [29] to keep track of scores associated with flipping more than one bit at a time, allowing an efficient way of identifying moves (combinations of a small number of concurrent bit flips) that will reduce the cost function (which is analogous to the energy function of a network).

2.5.2.1 *Local Minima*

Regardless of the descent route taken, the search will lead to a stable point from which no further improvement can be made. These points are known as local minima. One way to escape from local minima is to pick a new random starting point. The repetition of this approach, mixed with settling to an attractor from each new point is an approach known as random restart hill climb.

2.5.3 *Gibbs Sampling*

DEUM [117] searches for points (i.e. network states) with a high probabilty of being good solutions using Gibbs sampling, though this is presented as part of an evolutionary search. A similar MRF based approach, MARLEDA [2] uses Markov chain Monte Carlo sampling to generate good points. These methods can take time to produce samples as a 'burn in' period is required to allow the distribution of states visited by the sampling process to match that of the underlying distribution.

2.5.4 *Crossover Methods*

Whitley et al. [137] made use of a graph representation of a function, known as a variable interaction graph (VIG) to implement a genetic algorithm crossover method (called partition crossover). Two parents that are both local optima in the function can be recombined to produce a new candidate that is also a local optimum in a restricted hyperplane. This type of crossover is known as *respectful* as any bit shared by both parents is transmitted to the child. The job of a crossover operator is to decide which parent should transmit a value for each of the bits where they disagree. If there are v points of disagreement, there are 2^v possible offspring that can be produced. Partition crossover takes the VIG of the full function and removes the nodes and associated edges of variables with the same value in both parents to reveal a *recombination graph*. If this graph may be partitioned, then the subgraphs can be optimised in isolation as the contribution to energy of one is independent of the value of the other. This greatly reduces the search space and allows a new local optimum to be generated from the two parents.

Partition crossover relies on the VIG being known and amenable to partitioning, but when these conditions hold, it improves the performance of a GA with simpler 2-point or uniform crossover. The approach is presented as a crossover operator, not a graph energy minimisation method, but its purpose is the same.

The next chapter presents methods for inferring the right network for a search problem from samples of the fitness function and proposes a number of methods for searching the resulting network.

2.6 Summary

Systems of many binary variables can be feed forward, where the inputs, X map to an output Y or dynamic where the current values of X determine (perhaps stochastically) the values of X at the next discrete step. The second case is related to the first by an energy measure that is a function of network state.

The simplest feed forward function of many variables to a single output is the multiple linear model, which predicts the expected value of the distribution of the output variable, given the current input. Other distributions may be represented with the same model using a link function, leading to the generalised linear model. Using a logit link function and adding functions of hidden variables produces a multi layer perceptron, which is a universal approximator for feed forward functions.

Dynamic models of the type described here can be used either with deterministic dynamics, in which case they move from any state to either a fixed state or a cycle, or with stochastic dynamics where node states change according to a probability function. A network that follows a Boltzmann distribution has a probability of being in state x that is exponential in the negative of the energy of that state. Minima in the energy function (and, consequently, maxima of the Boltzmann distribution function) are described as attractors or memories in the model.

To model interactions between groups of more than two variables, networks may either make use of hidden units, such as those in a Boltzmann machine, or higher order connections such as those in a Markov random field. Discovering the correct structure of linkage among the input variables of a model is of prime importance and still an active field of research.

Both feed forward and dynamic models have been used to aid heuristic search for global (or sufficiently good local) optima in fitness functions. Such models may be used as surrogate fitness functions to speed the evaluation of candidate solutions, as representations of the constraints in the fitness function as an aid to satisfying those constraints by searching the model, or as an estimation of the distribution of good solutions from which new candidates for an evolutionary search are generated.

A good approach to building models from data and using them to predict or search for outputs should have a number of qualities. The representation should be powerful enough to capture any function required but the degree of complexity and the model bias that results in a lowering of that complexity should be readily controllable. Algorithms should exist for learning the parameter values of an existing structure and for discovering a good structure based on a sample of training data. Ideally, versions of the learning algorithm should also exist for directly setting attractors in the energy function without the need for learning the shape of the function away from those points. Further it can be useful to have versions of the learning algorithm that are both online and offline. It is also desirable that there are learning algorithms that can introduce model bias when the coefficients of a fixed model are learned.

Further desirable features of such a method are that the resulting model is human readable, which adds to comprehension and sense checking of the model but also allows comparison across an ensemble of models and the detection of local minima in the cost function. Human readability also allows prior knowledge to be injected by design. A good modelling approach should also be able to select input variables as part of its learning process and work even when variables make little contribution in isolation, their effect only being shown in their interactions.

If the model is to be used as part of a search method, there should be heuristics for finding and escaping local optima in the model and ideally those heuristics should be able to take advantage of the structure of the function represented by the model. The next section describes an approach to modelling that has all of the features described above.

Part II

CONTRIBUTION

3

MIXED ORDER HYPER NETWORKS

3.1 Introduction

This chapter introduces the mixed order hyper network (MOHN) and describes the technical details of building and using one. The chapter begins by defining the structure of a MOHN representation of a function as a hypernetwork with weighted hyperedges called weights. Calculations for mapping an input to an output are presented along with learning rules for estimating the parameters on the weights. A dynamic is then defined that allows the hypernetwork to move to local minima in an energy function. This dynamic is used to implement a content addressable memory in a MOHN. An algorithm for that attempts to discover the correct weights to include in a MOHN is then proposed and described.

The use of a MOHN as a fitness function model for heuristic optimisation is then discussed and several local search algorithms are presented, each of which makes use of the structure of the MOHN to some extent.

3.1.1 *Definition and Notation*

The notation used throughout this thesis is summarised in the preface. This section defines the data and functions under consideration and the structure of a mixed order hyper network.

3.1.1.1 *Data and Functions*

This work considers the use of a MOHN to represent a function $f : \{-1, 1\}^n \rightarrow \mathbb{R}$ based on data that reflects that function. The data may be generated in one of two ways. It may be collected using measurements of a real world process, in which case it is likely to be noisy. Alternatively the data may be generated from an existing function, in which case it is likely to be noise free. Samples may be input patterns alone or (input, output) pairs. Reasons for training a MOHN on data from an existing implementation of the target function include the testing of algorithms (most of the data in this work is generated artificially for this purpose) and the benefits of being able to search, analyse and visualise a MOHN, which the original function may lack. This section will continue to use the notation for input and output variables described in section 1.2.1.

3.1.1.2 *MOHN Structure*

A MOHN is a hypergraph, $\mathbf{M} = (X, \mathbf{W})$ where X is a vector of n nodes, which we shall call neurons, $X = X_1 \dots X_n$ and $\mathbf{W}$ is a set of weighted hyper-edges, each connecting zero or more of the neurons in X. Each weight[1] in $\mathbf{W}$ consists of a value, $\omega \in \mathbb{R}$ and a set of indices of the connected neurons, $\mathbf{I}$. A weight and its parts are indexed so $W_j = (\omega_j, \mathbf{I}_j)$. The neurons connected to W_j may be iterated over using $X_i : i \in \mathbf{I}_j$. The size of the set of connected neurons on a weight, $|\mathbf{I}|$ is known as the weight's order. The hypergraph is not directed, so each subset of nodes in a fully connected MOHN has a unique hyperedge and a single associated weight value.

The neurons map directly onto the input variables in the data, which is why both are denoted by the vector X. The function implemented by a MOHN is denoted $\hat{f}(X)$, given in equation 3.1 and the sharing of notation between variables and neurons allows that relationship to be written explicitly.

A lower case x is used to denote a particular instantiation of values across X, either the network's current *state* or an example training pattern. Let $\mathbf{W}^n$ represent the weights of a fully connected network. $\mathbf{W}^n$ contains 2^n weights. There is a single zero-order weight, which connects no neurons, but has a value all the same. There are n first order weights which are the equivalent of bias inputs in a standard neural network and $n(n-1)/2$ second order weights, the equivalent of those in a Hopfield network. In general, there are $\binom{n}{k}$ weights of order k in a fully connected network of size n. Most networks will have a sparse pattern of connectivity, so $\mathbf{W} \subset \mathbf{W}^n$ and $|\mathbf{W}| << 2^n$. Figure 3.1 shows a partially connected MOHN where $n = 4$ with the weight index sets shown.

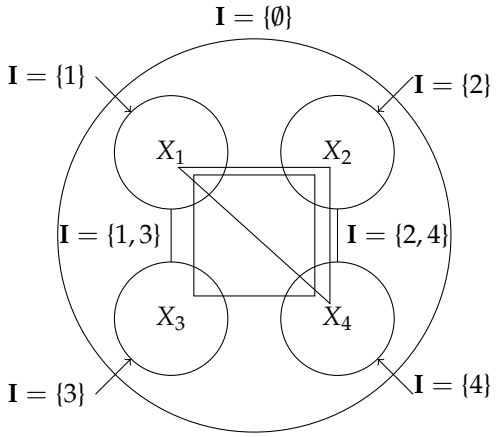

Figure 3.1: An example four neuron MOHN with sparse connections. The triangle has a connection set $\mathbf{I} = \{1, 2, 4\}$ and the square has $\mathbf{I} = \{1, 2, 3, 4\}$. The circles labelled X_1 to X_4 indicate the inputs and there is no explicit output node.

1 Strictly, weights are actually hyperweights, but the simpler term is used throughout this work.

3.1.1.3 *MOHN Function*

As stated in claim 3 from page 4, the MOHN function is a linear parameter model of the form

$$\hat{f}(X) = \sum_j \omega_j \prod_{i \in \mathbf{I}_j} X_i \tag{3.1}$$

Equation 3.1 is linear in the parameters, ω_j and can also be interpreted as a multivariate power series expansion without the X_i^m terms or as a basis over $f : \{-1, 1\}^n$ (though in practice many of the parameters will be zero valued).

3.1.1.4 *Energy Function*

A MOHN has an associated energy function, $U(X)$, which is defined in terms of the weights and the values on the neurons

$$U(X) = -\sum_j \omega_j \prod_{i \in \mathbf{I}_j} X_i = -\hat{f}(X) \tag{3.2}$$

The energy function of equation 3.2 is simply the negation of the MOHN function of equation 3.1. A weight defines a type of weak constraint among the neurons it connects. The constraint defines the desired sign for the product of values across the neurons connected by the weight. If the weight value is positive, then the product of values across the neurons it connects should be positive to satisfy the constraint defined by the weight. The magnitude of the weight defines its relative importance among all weights in the MOHN.

It is possible for weights in the same MOHN to define incompatible constraints such that no pattern of values across the inputs satisfies them all. This is why the constraints are known as soft. The function that a MOHN represents has its output globally maximised when its energy is globally minimised. This is at the point where the values across the neurons maximise the sum of the magnitudes of the weights whose constraints they satisfy.

Most of the remainder of this work deals with networks that are designed to learn a function from data, in which case equation 3.1 is used rather than its negation. Consequently, the task of maximising the function is discussed, rather than that of minimising the energy.

3.2 Learning Rules

This section presents the different methods for estimating the weight values of a MOHN from data. It assumes the structure of the MOHN, that is which weights are included, to be fixed. A method for discovering the correct structure from data is presented in section 3.3. The data used to train a MOHN may be either a noisy sample based on measurements from a real world

process or samples from a noiseless function. The former case is the usual machine learning or data mining scenario where a model is built from data. Learning a surrogate model of a fitness function for optimisation tasks is an example of the latter case. Either way, the data will consist of samples of the input patterns, x and (where available) an associated output value, y.

3.2.1 *Hebbian Learning*

The first learning rule we present is the only auto-associative rule, a high order extension to the Hebbian rule. In this case, each training example is an input vector, X and no function output is specified. Given that the inputs are set to a single pattern, $X = x$, that pattern is learned (added to the memory) by updating the weights using

$$\omega_j \leftarrow \omega_j + \frac{1}{m} \prod_{i \in \mathbf{I}_j} X_i \tag{3.3}$$

This is the outer product learning rule described by Venkatesht and Baldi [141]. With a zero diagonal in the weight matrix it allows the MOHN to be used as a content addressable memory (CAM). For a network that is fully connected at order two only, using equation 3.3 is the same as loading patterns into a standard Hopfield network. When the MOHN contains higher order weights, the capacity of the network is increased [141]. Patterns are recalled as they are in a Hopfield network, by setting the neuron values to a noisy or degraded pattern and allowing the network to settle using algorithm 9 on page 79. Up to the point where the capacity of the network is exceeded, a CAM trained in this way stores the training patterns as stable states but, with very high probability, also contains other spurious attractors.

3.2.2 *Weighted Hebbian Learning*

Kinser [76] describes a high order outer product rule in the context of mapping an input vector, X to a scalar output, Y. This rule multiplies each product in equation 3.3 by the output value, y associated with the input being learned. Kinser states that such a network has advantages over two layer first order networks but suffer from a limited ability to learn random high order problems. However, a high order version of the weighted outer product rule is capable of learning any arbitrary function in $f : \{-1, 1\}^n \rightarrow \mathbb{R}$. In fact, a fully connected MOHN forms a basis for functions in that space. To form the basis, the weight values are calculated from an exhaustive set of X, Y pairs using the Hebbian learning rule for each input, X, weighted by its associated output, Y.

$$\omega_j = \frac{1}{2^n} \sum_{X \in \{-1,1\}} f(X) \prod_{i \in \mathbf{I}_j} X_i \tag{3.4}$$

The weighted Hebbian weight calculation is very similar to the Walsh decomposition calculation given in equation 2.28 and reproduced below.

$$\omega_j = \frac{1}{2^n} \sum_{X \in \{1,-1\}^n} f(X)\psi_j(X) \tag{3.5}$$

where $\psi_j(X) = \oplus(X \wedge j_{bin})$ is a parity count of values set to 1 over the bits in X that are acted on by parameter ω_j. The equivalences to a fully connected MOHN are clear. Each ω_j corresponds to a single weight, $W_j = (w_j, I_j)$ where weight value w_j corresponds to the Walsh coefficient ω_j and the index set, I_j contains the indices of inputs that share a position in the input vector with the presence of a 1 in the same position in the binary representation of j.

The difference is that the output of $\psi_j(X)$ is the parity of the number of 1s in the chosen subvector of X and the equivalent in a MOHN is the product of the values across that same subvector, which amounts to the parity of the number of values set to -1. When the number of variables in the subvector is even, the parity of the number of 1s equals the parity of the number of -1s, so the MOHN and the Walsh functions are the same. When the subvector has an odd number of variables, the parities are different, so the Walsh function and the MOHN product differ in sign.

The weights of a fully connected MOHN trained using the weighted Hebbian learning rule of equation 3.4 on a single example of every possible (input, output) pair will produce a set of weights that are equal to the coefficients of a Walsh decomposition subject to

$$\omega_j = \phi(O(j))\omega_j \quad \forall j \tag{3.6}$$

where $\phi(O(j))$ is the parity of the order of ω_j such that:

$$\phi(O(j)) = \begin{cases} 1 & \text{if the order of } \omega_j \text{ is even} \\ -1 & \text{otherwise} \end{cases} \tag{3.7}$$

As the odd parity coefficients and the output of the functions both differ in sign from the MOHN to the Walsh domain, they are cancelled in the linear combination (equations 2.26 and 3.2) that calculates the function output, making their function identical.

This forms the proof of claim 1 on page 4, that by equivalence to the Walsh basis, a fully connected MOHN forms a basis for the functions in $f : \{-1, 1\}^n \to \mathbb{R}$.

Further more, as $X_i \in \{-1, 1\}$, the mean value $\bar{X}_i$ in an exhaustive and unique sampling of X will be zero as there will be an equal number of +1 and -1 values for X_i in the sample. Take the standard least squares estimator for the slope parameter, ω_1 in simple linear regression:

$$\omega_1 = \frac{\sum_{j=1}^{m}(X_j - \bar{X})(Y_j - \bar{Y})}{\sum_{j=1}^{m}(X_j - \bar{X})^2} \tag{3.8}$$

and let $\bar{X} = 0$ to obtain

$$\omega_1 = \frac{\sum_{j=1}^{m} X_j(Y_j - \bar{Y})}{n} = \frac{\sum_{j=1}^{m} X_j Y_j - \sum_{j=1}^{m} X_i \bar{Y}}{n} = \frac{\sum_{j=1}^{m} X_j Y_j}{n} \tag{3.9}$$

By the same process, the estimate for ω_0, the intercept, is

$$\omega_0 = \bar{Y} - \omega_1 \bar{X} = \bar{Y} \tag{3.10}$$

The weighted Hebbian parameter estimate is equal to the least squares estimate of a parameter in isolation when the distribution of input values is uniform across $\{-1, 1\}$ for each variable.

The Hebbian learning rule is interesting for three reasons. It forms the basis of the proof that a MOHN can represent any function in $f : \{-1, 1\}^n \rightarrow \mathbb{R}$ and it is used in heuristic search methods discussed in section 3.5.4. In addition, Swingler [133] has shown that a Hopfield network trained with the weighted Hebbian rule can learn the maximal turning points in a function from (X, Y) samples and has the same capacity as the same network trained using the Hebbian rule. The difference between the Hebbian and the weighted Hebbian approaches is that the Hebbian rule trains the network on known patterns (the attractors, or local maxima) and the weighted Hebbian rule discovers the location of the attractors from (X, Y) pairs, which do not need to include the local maximal patterns themselves.

3.2.2.1 *Parity Count Learning*

The weighted Hebbian learning rule estimates each parameter value independently and without accounting for any imbalance between the number of times the product across the connected inputs is positive or negative. Calculating the parameters independently introduces bias that can only be removed by considering all of the parameters together. However, gaining an initial biased estimate of the parameter values may be useful as a first step when estimating the parameters using a gradient descent approach (see section 3.2.3.3), so improving that estimate is of interest.

The consequence of any imbalance across the parity of the input patterns is that $\bar{X}$ in equation 3.8 is not zero and the estimate made by the weighted Hebbian rule is biased towards the values of Y associated with the most frequently occurring value of each X_i. This bias can be removed, while maintaining the practice of estimating the weight values independently, by applying equation 3.8 to each weight.

The same ratio in equation 3.8 is produced by replacing the denominator of equation 3.8 by the difference between the two possible input values, i.e. $1 - (-1) = 2$ and the numerator by the difference between the mean of Y when $X = 1$ and $X = -1$.

The sum in the weighted Hebbian rule of equation 3.4 adds values of Y that are associated with a positive product across a weight's inputs and subtracts values of Y that are associated

with a negative input product. All values are divided by n regardless of how often they appear. By splitting the sample into two groups, those with positive products across their inputs and those with negative, the two groups can be re-weighted. This is done by calculating the average value of Y in each group and allowing each resulting average to contribute equally to the estimate of the weight value by summing them and dividing by two.

Formally, $\mathbf{I}_j$ defines a subvector of X. Let $\mathbf{X}_j^+$ be the set of subvectors of each $x \in \mathbf{X}$ defined by $\mathbf{I}_j$ that contain values whose product is positive and $\mathbf{X}_j^-$ be the set of subvectors that contain values whose product is negative. Now let $\langle y^+ \rangle$ be the average value of y associated with the members of $\mathbf{X}_j^+$ and $\langle y^- \rangle$ be the average value of y associated with the members of $\mathbf{X}_j^-$:

$$\langle y^+ \rangle = \frac{1}{|X_j^+|} \sum_{x \in X_j^+} f(x) \tag{3.11}$$

Similarly, $\langle y^- \rangle$ is calculated as a sum over $x \in \mathbf{X}_j^-$. The weight value calculation for all weights except ω_0 is

$$\omega_j = \frac{1}{2}\Big(\langle y^+ \rangle - \langle y^- \rangle\Big) \tag{3.12}$$

The value associated with W_0, ω_0 is simply the average of Y across the whole data set. The parity learning rule is not of great interest in its own right, but will be considered again as an efficient method of finding a starting point for gradient descent learning in the next section. Section 4.3.6 describes an experiment in which the parity rule is used as an initialisation for a gradient descent error minimisation, for example.

3.2.3 *Regression Rules*

The weighted Hebbian learning rule treats the parameters to be estimated independently, which introduces bias when the assumption of uncorrelated inputs is violated. An unbiased estimate of the weight values can be found by minimising a squared error cost function over all the weights together. The squared error cost function, $C(\omega)$ maps the weight values, $\omega = \omega_1 \ldots \omega_j$, of a MOHN to the squared error on the training data that those weights produce.

$$C(\omega) = \sum_{x \in \mathbf{X}} \frac{1}{2}(f(x) - \hat{f}(x, \omega))^2 \tag{3.13}$$

where $\hat{f}(x, \omega)$ is the output of equation 3.1 in response to the input values, x when the weight values are set to ω.

A MOHN is a linear parameter model with inputs that are formed from the product of values on subsets of the input variables. Equation 3.1 makes that clear. Minimising the squared

error defined in equation 3.13 in a linear model when the errors are homoscedastic and serially uncorrelated produces an unbiased estimate of the model parameters. Additionally, the squared error cost function of equation 3.13 is convex, with a single global minimum.

3.2.3.1 *Ordinary Least Squares*

To use ordinary least squares (OLS) [52] to estimate the weights of a MOHN, an $m \times |W|$ matrix $\mathbf{X}$ is constructed where each row is constructed from a training example and each column represents a weight. The first column represents W_0 and always contains a 1. The remaining columns each represent a weight, W_j and contain the product of the values of the inputs connected by that weight, $\prod_{i \in \mathbf{I}_j} x_i$. A vector $\mathbf{Y}$ takes the output values associated with each of the input rows and the parameters are calculated using singular value decomposition:

$$\omega = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y} \tag{3.14}$$

where $\mathbf{X}^T$ is the transpose of $\mathbf{X}$, $\mathbf{X}^{-1}$ is the inverse of $\mathbf{X}$ and ω is a vector of weight values in the same sequence as the weights were inserted into the matrix $\mathbf{X}$.

3.2.3.2 *The Lasso Learning Rule*

It may be desirable to introduce estimation bias, particularly to impose a regularisation on the weights to attempt to avoid overfitting. The lasso [136] may be used to learn regularised values on the weights of the MOHN. Each input vector is set up in the same way as described for OLS, by calculating the product of the input values connected to each weight and the coefficients generated by the lasso are read back into the weights of the MOHN in the same order. The lasso is described in section 2.1.3.3. The fact that the lasso forces some weights to zero not only allows the lasso to reject input variables that contribute little, but also to reject higher order weights that are not needed. The lasso can be used as a simple method for choosing network structure by over-connecting a network and then removing all the zero valued weights after the lasso regression has been performed.

3.2.3.3 *Online Learning*

The weights of a MOHN can also be estimated online (where the data is streamed one pattern at a time, rather than being available in a matrix as in equation 3.14) using stochastic gradient descent (SGD). SGD is designed to minimise a cost function. In this case, the unregularised least squares cost function defined in equation 3.13 is used. It has a partial derivative with respect to weight ω_j of

$$\frac{dC}{d\omega_j} = \sum_{X \in \mathbf{X}} (f(X) - \hat{f}(X)) \prod_{i \in \mathbf{I}_j} X_i \tag{3.15}$$

Individual weights can be updated from individual samples, (X, Y) using stochastic gradient descent:

$$\omega_j \leftarrow \omega_j + \eta(f(X) - \hat{f}(X)) \prod_{i \in \mathbf{I}_j} X_i \tag{3.16}$$

where $0 < \eta < 1$ is the learning rate. The optimal learning rate may be sought experimentally and further improvements might be made using a dynamic learning rate [152].

The iterative nature of the algorithm allows for early stopping to be used to control for overfitting with reference to a validation set. To that end, the available training data is split into two sets. D_t is the training data and D_v is the validation data. Note that $D_t \cup D_v$ represents the full set of data to be used in the learning process and $D_t \cap D_v = \emptyset$. Other approaches such as k-fold cross validation could be used instead. A single member of D_t or D_v is a single (input, output) pair from the data, (x_j, y_j). Algorithm 2 describes the learning process.

Algorithm 2 Online MOHN Learning with Stochastic Gradient Descent

Let D_t be a subset of the available data to be used for training the network
Let D_v be a subset of the available data to be used for validating the network as it trains
Let $e = 0$ count the training epochs
for all $(x, y) \in D_t$ **do**
    Initialise the weights in the network using the parity rule of equation 3.12
end for
repeat
    for all $x \in D_t$ **do**
        Update the weights in the network using the SGD learning rule of equation 3.16
        Let C_t and C_v be the root mean squared errors that result from evaluating every member of D_t and D_v respectively
        Increment e
    end for
until C_t and C_v or e meet stopping criteria

Note that the weights are initialised with the parity learning rule of equation 3.16, not to random values as with an MLP. This is because there are no local minima in the error function and so no need for random starting points. In cases where the entire (input,output) space of the function may be noiselessly sampled, the initialisation step will produce the correct weights immediately, without the need for additional error descent learning. The learning algorithm will work without the initialisation (the weights can be set to zero) but there is experimental evidence that it requires more iterations of the learning cycle. See section 4.3.6 for an example of the improvement that this weight initialisation can bring.

3.2.3.4 *Stopping Criteria*

The squared error cost function is convex and has a unique minimum for a given training set. Algorithm 2 makes use of a validation set to allow early stopping to be applied, meaning that the algorithm is not allowed to find that minimum. The motivation behind this choice is that over fitting can occur if the training error is allowed to reduce too far but might be avoided by stopping the training process early. Prechelt [107] described three criteria for using the training and validation error for early stopping. They were generalisation loss, which measures the ratio of current validation error to the lowest observed so far, the quotient of generalisation loss and progress, which allows training to continue if the training error is still falling quickly, and sustained validation error increase. As none of these criteria can guarantee termination, a limit on the number of epochs to train and a target error progress are also used. As discussed in the introduction, more sophisticated approaches to choosing the right model such as cross-validation are often used.

Algorithm 2 uses RMSE as part of the stopping criteria as that is the cost that is being minimised, but it can be useful to use the correlation between the MOHN output and the target outputs in the validation data, D_v as the measure that is reported as training progresses. Correlation is not dependent on the scale of Y like RMSE is, so can be more informative to the human observer.

3.2.4 *Comparing Learning Rules*

Experimental comparisons of the learning rules' efficiency and accuracy will be presented in the next chapter, but some comparisons can be made at this point. SGD has the advantages of requiring less memory than OLS as the data does not need to be stored in a matrix. When data sets are very large, this may become particularly important. It also has the advantage of being what De Campos et. al [38] call an anytime solution as the error diminishes gradually with time rather than needing the full matrix calculation of OLS to complete. When data sets are very large, such approaches have the additional advantage of possibly not even needing to use all of the available data as the stopping criteria may be met before all of the data has been processed once.

OLS provides an unbiased estimate of the weight values, whereas the lasso offers control over the degree of estimation bias. In this work, SGD relies on early stopping to introduce estimation bias, but other methods such as adding noise to the training data may also be used. The next section describes methods for controlling model bias by adding and removing weights from a MOHN in an iterative process that requires the weights to be re-estimated at each iteration. SGD is able to continue to learn the weights of each new structure using the values from the

previous iteration as starting points. See figure 4.40 for an example of SGD learning during structure discovery.

3.3 Structure Discovery

Section 1.3 states a number of properties of MOHNs. They form a basis for real valued functions of binary vector inputs, but the basis model contains 2^n weights, where n is the size of the input vector. For some functions, many of the parameter values are zero, which means the weights have no effect on the function output and can be removed. We call these *sparse* functions. MOHNs are linear parameter models so estimating the weight values requires at least as many data points as there are parameters in the model. Reducing the number of parameters that are estimated also reduces the required size of the data set. When modelling a fitness function from noise free samples, the number of function evaluations required is determined by the number of parameters in the MOHN.

It is trivial to show that sparse functions exist. Let $n = 2$ and $f(X) = 3X_1 - 2X_2$. In this example, $f(X)$ is sparse because the parameter associated with X_1X_2 is zero. Multiple linear regression on the input variables alone (without considering any interactions) creates a sparse model and makes the assumption that the target function that generated the data is also sparse. Almost all of the work in this thesis makes an assumption of sparsity about the functions being modelled. A method for discovering which parameters are non-zero within the constraints of small data samples is needed.

Section 2.4 describes a number of methods for choosing which parameters to include in a statistical model. We call this *structure discovery*. Algorithm 1 on page 45 proposes a generic approach to structure discovery that summarises a number of approaches from the literature. Many existing methods reduce the search space by defining a subset of model structures to explore. Some employ a greedy approach that requires large numbers of candidate parameters to be considered at each iteration. We believe that a new method is required, which is capable of providing every possible structure a non-zero probability of being chosen without the need for exhaustive consideration of each candidate parameter at each iteration.

The number of weights in a MOHN must be smaller than the number of training examples available. Limits on data quantity, either due to limitations on collection or by a desire to minimise the number of fitness function evaluations, place limits on model size. If the number of weights to consider is larger than the training set size, a method of adding and removing weights is required. Removing weights is the easier of the two. Many methods such as those reviewed in section 2.1.2 have been proposed. The task of choosing which weights to add is more challenging.

Greedy methods add a weight at each iteration, considering all candidate weights. For MOHNs of even moderate size, it is impossible to test every possible weight, even in isolation, so a method for choosing which weights to consider is needed. There are a number of requirements for such an algorithm. It must only consider a small proportion of all the possible weights but be capable of exploring a number of weights that is larger than the size of the training data. It should be possible for a user to introduce domain knowledge about the function if available. The algorithm should not rely purely on low order interactions to detect higher order ones, for example it should be able to discover the correct weights for a function that has connections at order three alone. It is also desirable to minimise the number of training examples the algorithm requires.

The goal of the structure discovery algorithm is to minimise a cost function that measures both the accuracy of predictions and the number of weights used. This suggests an L_1 regularisation approach, as discussed in section 2.4.4, which leads to a sparse connection structure. However, there is an additional constraint on the number of weights that can be compared at any one time, which is determined by the number of training examples available and the level of noise in the data. This motivates an iterative approach to adding and removing weights that keeps the current model size within bounds dictated by the data set.

Many greedy approaches, such as [106] evaluate every available feature and add only one at a time. For large networks, exhaustively considering each unused weight is impossible, so a smaller candidate set must be maintained. Adding or removing weights requires the weight values to all be re-estimated, so it is also desirable to add and remove as many weights as possible at each step, rather than the one-at-a-time approach of many greedy algorithms. A method of fitting the weight values at each iteration efficiently is also needed.

3.3.1 *The MOHN Structure Discovery Algorithm (MSDA)*

This section proposes and describes the MOHN structure discovery algorithm (MSDA). The algorithm is incremental, so weights are added and removed as it progresses. Regularisation is applied by the choice of weights to add or remove, but can also be introduced into the regression algorithm used to learn the weight values. When we refer to adding or removing a weight, or recording a set of used weights, the *weight* refers to the subset of neurons it connects, $\mathbf{I}_j$, not its value, ω.

The MSDA maintains a probability distribution, $P(\mathbf{I})$ from which candidate weights are sampled and added to the model. The model then undergoes a training phase after which all the weights are tested for significance. Insignificant weights are removed and as the model grows, the weight picking distribution is altered to reflect its emerging structure. This approach is similar to the probabilistic approach taken by Kim and Zhang [75] and to the same

authors' Bayesian evolutionary approach to classification [120], but it does not apply a Bayesian framework and is not designed to build classifiers.

Algorithm 1 on page 45 outlines a general algorithm for network structure discovery. The proposed MSDA is presented below with reference to algorithm 1. At its most abstract level, the algorithm proceeds as follows:

Algorithm 3 MOHN Structure Discovery Algorithm (MSDA).

Let $\mathbf{M} \leftarrow \emptyset$ be the set of weights in the current model
Let $\mathbf{H} \leftarrow \mathbf{I}^n$ be the full set of possible weights
Initialise a discrete distribution, $P(\mathbf{I})$ for $\mathbf{I} \in \mathbf{H}$
repeat
  Sample some weights, $\mathbf{C}$ from $\mathbf{H}$, each with probability $P(\mathbf{I})$ without replacement
  Add $\mathbf{C}$ to $\mathbf{M}$
  Remove $\mathbf{C}$ from $\mathbf{H}$
  Calculate the weight values for the resulting network, $\mathbf{M}$
  Regularise by removing some weights from $\mathbf{M}$
  Update the weights distribution, $P(\mathbf{I})$ to reflect what has been learned
until Stopping criteria are met

A number of decisions are required when implementing algorithm 3 in detail. They are:

- A representation of the probability mass function, $P(\mathbf{I})$ over unpicked weights
- A method for updating $P(\mathbf{I})$
- A choice of learning rule for calculating the new weight values in $\mathbf{M}$
- A choice of regularisation method for removing weights from $\mathbf{M}$

The following sections consider these points in more detail. These sections compare a number of choices for each step and are followed by an example algorithm based on one choice from each.

3.3.2 *Representing the Probability Distribution Across Weights*

The structure discovery algorithm is based on the premise that as not all possible weights can be considered, heuristics for picking weights that have a higher chance of proving useful must be used. The solution is to maintain a probability mass function over the possible weights where the probability of a weight being selected is proportional to its chance of being useful. This requires a representation of the space of possible weights and a method for shaping a function to reflect a weight's potential usefulness.

In algorithm 3, $\mathbf{H}$ represents the set of all possible weights and grows exponentially in size with n so it is not feasible to assign a probability to each weight in $\mathbf{H}$. Rather than use a single distribution that covers every possible weight, in this work two distributions are used. One covers the order of the weight and the other covers the probability of each neuron in the network being connected to that weight. Let the probability distribution over the weight orders, o of an n neuron MOHN be

$$Po(o) : o \in \{1, \ldots, n\} \tag{3.17}$$

and the probability of picking neuron X_i to be connected by the current choice of order o weight be

$$Pn(i) : i \in \{1, \ldots, n\} \tag{3.18}$$

The order, o is sampled first, and then a subset, Q of o neurons are sampled without replacement from $Pn(i)$. Both distributions are discrete—there are n possible orders and n possible neurons to choose from—so their representation need not be from any parametrised class. The probabilities can be represented as a vector of size n with the usual constraint that each must be between 0 and 1 and they must sum to 1.

For a given weight, $\mathbf{I}$ of order $o = |\mathbf{I}|$, the probability of being selected by the algorithm, $P(\mathbf{I})$ is

$$P(\mathbf{I}) = Po(o) \prod_{i \in \mathbf{I}} Pn(i) \tag{3.19}$$

There are 2^n possible weights but the size of $Po(o)$ and $Pn(i)$ are both n, making the number of discrete probabilities that are calculated only $2n$, making the maintenance of the distributions linear in n. How $Po(o)$ and $Pn(i)$ evolve as the algorithm progresses is addressed next.

3.3.3 *Updating the Weight Picking Distributions*

At the first iteration of the algorithm, the distributions $Po(o)$ and $Pn(i)$ must be set up manually. This presents an opportunity to include any prior knowledge that exists about the function to be modelled and also allows some control over the complexity of the model to be imposed.

3.3.3.1 *Distribution over Weight Orders*

The initial distribution over the weight orders needs to be defined over the integers between 1 and n and sum to one over that range. It should also allow a very tight concentration at a single weight order if required, so fall from the mode exponentially with distance from it. The discrete

Laplace function with support over $\{1, \ldots, n\}$ has all of these properties. The discrete Laplace, which is a discrete analogue of the continuous Laplace distribution is defined as

$$f_L(x) = \frac{1}{2\lambda} e^{-\frac{|c-x|}{\lambda}} \tag{3.20}$$

where λ controls the width of the distribution and c defines the mode. The discrete Laplace has a probability mass function defined by

$$Po(o) = \frac{f_L(o)}{\sum_{m=1}^{n} f_L(m)} \; o \in \{1, \ldots, n\} \tag{3.21}$$

In the early iterations of the algorithm where $c = 1$, there is a high probability of picking first order weights and an exponentially decreasing probability of picking weights of higher order. In subsequent iterations, $Po(o)$ is updated in two ways. Firstly, c is increased to allow the algorithm to pick weights with higher orders and secondly the values of existing weights are used to shape the distribution to guide the algorithm towards orders that have yielded high value weights already. Note that the number of available weights at each order is not taken into account in this process.

After some weights have been removed, the weight order probability distribution, $Po(o)$ is updated so that each order's probability changes according to the contribution to the total sum of absolute weight values made by weights at each order. Let S be the sum of the absolute weight values across the network, $S = \sum_j |\omega_j|$ and

$$R_o = \frac{1}{S} \sum_{j:|\mathbf{I}_j|=o} |\omega_j| \tag{3.22}$$

be the proportion of S accounted for by weights at order o. These proportions are then used to update $Po()$ along with an updated version of the discrete Laplace distribution as follows:

$$Po(i) \leftarrow (1 - (\alpha + \beta))Po(i) + \alpha R_i + \beta \frac{1}{2\lambda} e^{-\frac{|c-o|}{\lambda}} \tag{3.23}$$

where α is the proportion of R_o to include in the update and β is the proportion of the current order mode, c that is included such that $0 \leq \alpha \leq 1$, $0 \leq \beta \leq 1$ and $0 < \alpha + \beta \leq 1$.

If $\alpha + \beta = 1$ the new distribution is a mixture of the current distribution of weight orders in the MOHN and the discrete Laplace distribution with a mode of c. If $\alpha + \beta < 1$ the distribution retains some memory of its previous shape, weighted by $1 - (\alpha + \beta)$. In the experiments reported in this paper, the values $\alpha = 0.6$, $\beta = 0.2$ were used and found to work well.

The weight order mode, c needs to be manipulated as learning progresses. In the work reported here, c was set to equal the lowest order with remaining unsampled weights. As lower weight orders are exhausted, the mode naturally moves up. Of course, this does not rule out

higher rates being sampled — the α component will bias the sampling towards higher orders if they prove useful. The smaller the value of λ, the faster the weight order distribution drops towards zero as it moves away from c.

3.3.4 *Distribution over Neurons*

Once the order, o of a new candidate weight has been sampled, the o neurons that it connects must be picked. These neurons are picked from a distribution, $Pn()$ that evolves as each neuron is picked. The shape of $Pn()$ is determined by a number of factors. Prior knowledge can be included by increasing the probability of variables that are known to be useful. If no prior knowledge is available, then $Pn()$ starts off as a uniform distribution. Once there are some weights in the network, $Pn()$ is determined by a mixture of the prior knowledge and the role played by each neuron in the existing network. To connect a weight of order o, there are two phases to the neuron picking procedure. The distribution from which the first neuron is picked is shaped by the contribution each neuron is already making. In exploratory mode, neurons that have not yet played a role are favoured and in exploitative mode, neurons that are already well connected are more likely to be picked. Subsequent neurons, up to a count of o, are picked from a distribution that is reshaped by the set of neurons that are already connected to the existing set under construction at orders other than o.

The trade-off between exploration and exploitation can be managed. Exploration in this case means favouring neurons that have few or weak connections on the assumption that they do have a role to play, but it has yet to be found. Exploitation refers to picking neurons that already have connections on the assumption that those which have proved useful at some orders will also be useful at others.

The first step in picking the o neurons is to pick the first with a probability proportional to the contribution it makes to the model. Define the contribution made by neuron i as being the sum of the absolute values of the weights connected to neuron i.

$$C(i) = \sum_{j:i\in\mathbf{I}_j} |\omega_j| \tag{3.24}$$

where $j : i \in \mathbf{I}_j$ iterates over the values of the weights connected to X_i. The proportion of the total contribution of all neurons made by neuron i is

$$Cp(i) = \frac{C(i)}{\sum_{k=1}^{n} C(k)} \tag{3.25}$$

Now let ρ control the level of exploration, such that $\rho = -1$ means full exploration (bias the search towards unused neurons), $\rho = 1$ means full exploitation (bias the search towards well used neurons) and $\rho = 0$ leads to a uniformly random choice among the neurons. Any

other value of $-1 < \rho < 1$ balances the degree to which exploration or exploitation is made. When $\rho > 0$, the contribution made by its connectivity strength, $C(i)$ is the proportion $Cp(i)$. However, when exploring, with $\rho < 0$, the contributions need to be reversed so that the neuron with the maximal value of $Cp(i)$ has the smallest probability of being picked, the neuron with the smallest $Cp(i)$ is most likely to be picked, and those in between are linearly transformed in between. To achieve this, let

$$R(i) = \max(Cp) + \min(Cp) - Cp(i) \tag{3.26}$$

and let the exploration proportion for neuron i, $Rp(i)$ be

$$Rp(i) = \frac{R(i)}{\sum_{k=1}^{n} R(k)} \tag{3.27}$$

The probability of picking neuron i is

$$Pn(i) = \begin{cases} (1-\rho)\frac{1}{n} + \rho Cp(i), & \text{if } \rho > 0 \\ (1+\rho)\frac{1}{n} - \rho Rp(i), & \text{otherwise} \end{cases} \tag{3.28}$$

Equation 3.28 causes the degree of exploration to vary when $\rho < 0$ and causes the degree of exploitation to vary when $\rho > 0$. The closer to zero the value of ρ gets, the more uniformly random the neuron selection becomes.

Defining a weight involves building the weight's connection index set, $\mathbf{I}$. Each neuron connected by the weight being built is selected by sampling an index from $Pn(j)$ without replacement. Once a neuron, X_i is picked, $Pn(j)$, $j \notin \mathbf{I}$ are updated in two ways. Firstly, the chosen neuron has its probability set to zero to prevent it being picked again and the probabilities of the remaining available neurons (those not in $\mathbf{I}$) are increased by an equal amount to force them to sum to one.

Then, $Pn(j)$, $j \notin \mathbf{I}$ are updated again so that other neurons that are already connected to X_i at other orders have their probability of being chosen increased, while neurons that are not connected to X_i at other orders have their probability decreased.

Each neuron's new probability, $Pn(j)$ is updated so that it receives a proportion of its current value and a proportion weighted by its connectivity to other neurons already on the weight under construction. Let V be the set of weights that are connected to any of the neurons that have been picked for the weight currently under construction. The contribution of a candidate neuron, X_j, based on its connectivity to the last picked neuron, X_i is $Cu(j,i)$ and defined as

$$Cu(j,i) \sum_{k:W_k \in V, X_i \in \mathbf{I}_k} |\omega_k| \tag{3.29}$$

Any neuron that is not currently connected to any of those currently on the weight being constructed have $Cu = 0$. These values are then normalised so that they sum to one:

$$Cn(j) = \frac{Cu(j)}{\sum_{k=1}^{n} Cu(k)} \tag{3.30}$$

The sum is over all weights that are connected to both X_i and any of the other neurons already chosen for the new weight. Each probability, $j \neq i$ is then updated with reference to i as follows

$$Pn(j) \leftarrow (1-\delta)Pn(j) + \delta Cn(j,i) \tag{3.31}$$

The parameter $\delta \in [0,1]$ controls the mix of the previous shape of $Pn()$ and the update. High values of δ cause the algorithm to favour neurons that are connected to those already in the set being built, and low values cause it to favour the contribution of each neuron in isolation. In this way sets of neurons that form cliques due to low order connections have a higher probability of being connected at higher orders. Finally, when the number of neurons picked equals $o-1$, the probability associated with all neurons already connected to those neurons at order o is set to zero to ensure an existing weight is not picked.

Unless the inputs to all pairs of weights are uncorrelated, when weights are learned in isolation, bias is introduced. That suggests that learning many weights at once is desirable. There should not be more weights in a model than there are training data points used to estimate their values, so the number of weights added should be chosen to ensure that the model has no more weights than there are training examples.

3.3.4.1 *Efficient Weight Picking*

Once a weight is already in the model or has been tested and discarded, it is considered *used*. Only unused weights should be considered for addition to the model. When the ratio of available weights to used weights is high, it is efficient to simply pick a random weight using the procedure above and check that it is not already in the network or in a list of weights that have been considered but removed from the network. To avoid unuseful weights being repeatedly added and removed, a list of discarded weights is maintained. Newly sampled prospective weights are first compared to the members of this list and not added if they have been recently tried. As the model is not invariant under reparametrization, weights may appear unuseful as part of a poorly structured network, but later prove to be of use when the rest of the structure is in place. The discard list is periodically emptied to allow weights another chance of inclusion.

This approach becomes inefficient when there are very few weights available at the chosen order, meaning very many choices are required before an available weight is found. To ensure that there are available weights at the chosen order, the algorithm keeps count of how many

weights of each order have been used. There are $\binom{n}{o}$ possible weights at each order, o, so when the order o count reaches this figure, the probability of picking a weight at that order is forced to zero.

Another efficiency enhancement to the algorithm is the inclusion of a 'mopping up' procedure that is activated when the number of used weights at order o reaches a certain percentage of the total (a threshold of 90% is used in this work). When the order o count reaches the threshold, the few remaining weights at order o are automatically added to the model and assessed. This allows the probability of picking from order o to then be forced to zero, thus avoiding many fruitless picks from that order.

Algorithm 4 Algorithm for picking a new set of weights to add to an existing MOHN

Let t be the number of weights to add
Let **U** be the set of discarded weights
Let **M** be the current network weights
Let $\mathbf{V} = \emptyset$ be a new weight set
Let $Po()$ be the probability distribution over the possible weight orders
Let $Pn()$ be the probability distribution over the possible neurons
repeat
  Let W_{new} be the new weight being built
  Pick an order, o from $Po()$
  if $\mathbf{U} \cup \mathbf{M} \cup \mathbf{V}$ is more than 90% full at order o **then**
    Add the rest of the unused order o weights to **V**
  else
    Set an initial distribution across the neurons, $Pn()$
    repeat
      Update $Pn()$ according to current network structure using equation 3.28
      Choose a new neuron X_i from $Pn()$
      Add X_i to the neuron set connected by W_{new}
      Update $Pn()$ based on the connectivity of X_i using equation 3.31
    until o neurons have been selected
    if $W_{new} \notin \mathbf{U} \cup \mathbf{M} \cup \mathbf{V}$ **then**
      Add W_{new} to **V**
    end if
  end if
until $|\mathbf{V}| >= t$

3.3.5 *Learning Rules for the Weights*

Section 3.2 summarised three regression learning rules for a fixed structured MOHN: SGD, OLS and the lasso. Each method has different advantages and disadvantages for estimating weight values during structure discovery. At each iteration of the structure discovery algorithm, a small proportion of new weights are added to a network whose existing weight values are likely to already be close to the correct value. As SGD is incremental, it can take advantage of this fact rather than starting a new, empty network at each iteration. New candidate weights can be initially set using equation 3.7, after which the entire new network is improved by SGD using equation 3.16. Algorithm 5 describes this process.

The nature of the regularisation in the lasso means that weights that are not needed have values forced to zero, removing the need for an additional weight removal decision, but at the cost of estimating the entire network structure from scratch at each iteration. The lasso also imposes regularisation on the weight values, which may or may not be desirable. The penalty applied by the lasso corresponds to a Laplace prior over the weight values. This expects many weight values to be at or close to zero and only a few of them to be large, which corresponds well to the assumption made about the weights in a MOHN. In some experiments in this work, the lasso has been used to discover network structure, but the final weight values (after removing those with parameter values of zero) are then re-estimated using OLS to remove the estimation bias that lasso introduces. This is done in cases where the true, unbiased weight values can be derived from knowledge of the model and the result of the MSDA needs to be compared to those known values. Algorithm 6 describes the lasso network update method. A single value for λ may be chosen or, as is usual in the application of the lasso, a number of different settings for λ may be tried using least angle regression [41] (see section 2.1.3.3).

Algorithm 5 Weight update algorithm for SGD learning

Let $\mathbf{W}$ be the current network weights

Let $\mathbf{V}$ be a set of new weights, chosen using algorithm 4

Initialise the weights in $\mathbf{V}$ using the parity calculation, equation 3.7

Add the weights in $\mathbf{V}$ to $\mathbf{W}$ so $\mathbf{W} \leftarrow \mathbf{W} \cup \mathbf{V}$

Run SGD learning, equation 3.16 on $\mathbf{W}$

Algorithm 6 Weight Update Algorithm for Lasso learning

Let $\mathbf{W}$ be the current network weights

Let $\mathbf{V}$ be a set of new weights, chosen with algorithm 4

Add the weights in $\mathbf{V}$ to $\mathbf{W}$ so $\mathbf{W} = \mathbf{W} \cup \mathbf{V}$

Estimate the weight values using the lasso based on equation 2.13

3.3.6 *Regularisation and Weight Removal*

Regularisation refers to the process of introducing additional constraints to a machine learning process to prevent over fitting. This often takes the form of a penalty on complexity or a bound on the norm of the learned parameters. Regularisation can also involve the use of an out of sample validation set. All of these methods may be applied to a MOHN but the main means of regularising a MOHN is the removal of insignificant weights. In this section, two options for weight removal are considered. It is important to remove weights because the rules for updating the probability distributions from which new weights are chosen depend on the presence or absence of weights in the model. It is also desirable to keep the model small for reasons of parsimony, to avoid over fitting and to reduce the time and data quantity required during learning.

Equation 3.7 shows a first approximation to the correct value of a weight based on the difference between the mean function output for even and odd parity inputs to the weight. In cases where the difference between the distributions of the function output under each of the two parity conditions is not statistically significant, the weight may be excluded. A t-test is used to compare the mean function output between the odd and even parity input sets, allowing weights with a p-value above a chosen threshold to be removed. Some fine tuning of the critical p-value (*pcrit*) is required to ensure that the algorithm does not discard too many or too few weights. This can be achieved by trial and error or by including *pcrit* as one of the hyperparameters in a grid search. The t-value, t_j associated with weight W_j is calculated as

$$t_j = \frac{\omega_j}{\sqrt{\frac{\sigma}{m}}} \tag{3.32}$$

where ω_j is the weight value, σ is the variance of $f(X)$ and m is the number of training data points. The t-test is used as an approximation to the z-score calculation given in equation 2.14, without the need to calculate $(\mathbf{X}^T\mathbf{X})^{-1}$. The t-test makes the assumption that the difference between the average function output when the input to a weight is positive and that when it is negative is normally distributed. An alternative, the Mann Whitney test may be used when this assumption does not hold. The experiments reported in this work all use the t-test, however.

Section 3.3.5 describes the lasso approach to estimating the weight values, including the fact that the regularisation forces parameter values towards zero, making the choice of which weights to remove from the network straight forward.

The least squares estimate of the value of any parameter, ω_j depends to some extent on which other weights, $W_{i\neq j}$ are included in the model and the values they take. Consequently, whether or not ω_j should be included in the model also depends on the other weights. Experimental evidence suggests that some weights that have significant values when certain other weights are present can be judged insignificant when those weights are not present in the model. For

this reason, weights that are removed during early cycles should be given another chance of inclusion later in the learning process. This is done by emptying the used weight set, **U** according to some schedule. The schedule may be chosen based on the number of elapsed epochs. In some of the experiments reported in this work, the schedule was chosen by first running the learning algorithm without emptying **U**. The number of epochs taken before the validation error shows little change from epoch to epoch is then used as the interval for emptying **U**. A second approach, which was also used here was to include the reset schedule as a hyperparameter in a grid search approach to finding the best model.

3.3.7 *The Full Algorithm*

The full structure discovery algorithm is presented in algorithm 7, with reference to partial algorithms already described above.

Algorithm 7 Full MOHN Structure Discovery Algorithm.

Start with an empty network, $W = \emptyset$
Initialise an empty used weight set, $\mathbf{U} = \emptyset$
Initialise the probability distribution $Po()$ over the possible network orders $o = 1 \dots n$
repeat
  Reset $\mathbf{U} = \emptyset$ according to a schedule
  Use algorithm 4 to select a new set of candidate weights, **V**
  Train and merge **V** and **W** using either algorithm 5 or 6
  Remove weights based on either a t-test or the zero valued weights after the lasso
  Add the removed weights to **U**
  Recalculate $Po()$ using equation 3.23
  Calculate the validation error
  Update the parameters c and $pcrit$
until The validation error is sufficiently low or no longer improves

3.3.8 *Structure Discovery for Content Addressable Memories*

An adaption is now proposed to Algorithm 7 that allows it to discover a set of weights capable of storing a set of patterns in a content addressable memory (CAM). The high order CAMs discussed so far in this work [81] [140] are fully connected, so the number of weights grows rapidly as orders are added. Discovering a sparse representation allows an arbitrary pattern set to be represented in a CAM with far fewer weights. The disadvantage of this approach is that each time a new pattern is learned, all of the previous patterns must also be re-learned as a new structure is found.

The t-test for statistical significance of weights in a content addressable memory is not appropriate as there is no output, Y. The distribution of parity values across a weight determine whether or not the weight is useful. A uniform distribution suggests the weight is not useful. This suggests Pearson's Chi-squared test with the null hypothesis that the number of patterns with an even parity across a weight is equal to the number with an odd parity. Weights that satisfy the null hypothesis are removed.

Let $\mathbf{X}$ be a set of patterns to be stored in a CAM. Let P_j^+ be the number of patterns in $\mathbf{X}$ that have an even parity over the subvector associated with weight W_j and P_j^- be the equivalent count of odd parity patterns. The Chi-squared test statistic for weight j is

$$\chi_j^2 = \frac{(P_j^+ - \frac{|\mathbf{X}|}{2})^2 + (P_j^- - \frac{|\mathbf{X}|}{2})^2}{|\mathbf{X}|} \tag{3.33}$$

where $|\mathbf{X}|$ is the number of patterns to learn and $P_j^+ + P_j^- = |\mathbf{X}|$ for all j. The result, χ_j^2 is tested for significance against the chi squared distribution with 1 degree of freedom. Algorithm 8 summarises the CAM structure discovery algorithm, which follows the same pattern as MSDA and makes use of most of the same processes. The main differences are that learning is Hebbian rather than using the lasso or SGD and the significance test is Chi-squared.

Algorithm 8 Structure discovery algorithm for content addressable memory.

Let $\mathbf{X}$ be the set of patterns to store
Start with an empty network, M
Initialise the probability distribution $Po()$
repeat
  Choose new weights to add to the network as follows:
  Pick an order, o from $Po()$
  repeat
    Set $Pn()$ according to current network structure using equation 3.28
    Choose a new neuron X_i from $Pn()$
    Add X_i to the set connected by the new weight
    Update $Pn()$ based on the connectivity of X_i using equation 3.31
  until o neurons have been selected
  Learn every pattern in $\mathbf{X}$ using the standard Hebbian learning rule
  Remove any insignificant weights using a Chi-squared test
  Recalculate $Po()$ using equation 3.23
until Every pattern in $\mathbf{X}$ is a stable attractor in M

3.3.9 *Monitoring the Learning Process*

As the algorithm progresses, the number of weights of each order in the network may be reported and compared to the possible total. The maximum number of possible weights at each order is $\binom{n}{o}$ where o is the order and n is the total number of inputs. This gives a measure of the complexity of the network compared to possible complexity. By reporting the list of tried and discarded weights, it is also possible to monitor how much of the weight space the algorithm has sampled.

To ensure the cost function is convex and not under specified, the current number of weights in the MOHN during learning must be kept smaller than the number of training examples. In cases where data is generated from a function, the size of the sample may grow in response to the size of the MOHN as it grows.

3.3.10 *Setting the Hyperparameters*

The MSDA has a number of hyperparameters, which control the way in which the algorithm works and encode some assumptions made by the user. They are listed below with comments on choosing suitable values based largely on experience gained from running experiments with the MSDA. We describe two learning methods, which we refer to as SGD and lasso. In the context of this work, we refer to minimising squared error, but with early stopping as the SGD approach and minimising the L1 cost function as the lasso approach.

- **Learning method**: Lasso has the advantage of forcing some weight values to zero, making the decision of which to remove straight forward. Stochastic gradient descent has the advantage that the values on the weights that remain in the model provide a very good starting point for the next iteration of the algorithm after weight removal and addition. On large data sets, this has been found to offer a considerable speed up in learning. We suggest that OLS is not a suitable choice as weight values need to be re-calculated at each iteration, rather than continuing from the current point using SGD. Experimental evidence for this is presented in section 4.4.3.
  - **Stochastic Gradient Descent**
    - **Number of epochs to train**: During gradient descent, the MOHN weights are updated across several passes through the training data (so called training epochs). Each iteration of the MSDA involves a number of training epochs on the current model. How many are required depends on a number of things, such as the size of the training data set. We suggest stopping when the validation error ceases to significantly fall. This can be identified by setting a minimum average decrease size over a number of epochs. Finding a good value can take

some experimentation. Plotting error curves over time can help the user choose a threshold or a fixed number of epochs.

* **Critical p-value for weight removal**: We have found that the p-value used for discarding weights is one of the harder hyperparameters to set. A value that neither discards nor keeps too many weights can be hard to identify without experimentation. Also, as the quality of a model improves, the critical value should reduce to account for the improved accuracy of the estimated weight values. This is a good hyperparameter to include in a grid search. Alternatively, starting high (0.5 has been found sufficient in most cases) and reducing by 0.05 every time the weight removal step fails to remove a weight has proved to be a useful heuristic.

– **Lasso**

* **Degree of regularisation**: The regularisation term λ in the lasso cost function determines the degree of regularisation and also influences the number of weights that take a zero value. The best level of regularisation can be explored by including λ in the grid search. We have found that low levels of regularisation worked well and higher levels tended to leave too many parameters with zero values.

- **Number of weights to add at each iteration** The experiments we report in this work allow for an initial number of weights to be introduced for the first iteration of the MSDA and for subsequent iterations to have another (usually smaller) number of weights added. It is essential to keep the number of weights lower than the number of training points in the data.

- **Initial distribution of weight orders** In the absence of domain knowledge, a sensible default assumption is that low order weights are sufficient. In all the examples given in this thesis, we have used the discrete Laplace distribution centred on order 1 with $\lambda = 1$, which causes 95% of the distribution to fall within two steps of the mode. The space of potential distributions is too large to address with a grid search and this hyperparameter should not be considered as part of such a search.

- **Update rate of weight order distribution** As evidence of useful weights is found, the order distribution is updated. The distribution itself also changes over time according to a pre-defined schedule. The schedule used in all of this work is to place the mode of the distribution over the first weight order that has not been exhaustively searched. The contribution made by new evidence to each update is controlled by the hyperparameter α and the proportion contributed by the distribution update schedule by β. We have fixed these values at $\alpha = 0.6, \beta = 0.2$ for the experiments in this work.

Hyperparameter	Values
Learning method	Lasso or SGD
Exploration trade-off	$\delta \in [0, 1]$
Weight order distribution update	$(\alpha, \beta)\{(0.6, 0.2), (0.5, 0.5), (0.2, 0.6)\}$
Number of weights added per iteration	$A \in \{n, 2n, 4n, 10n, 20n\}, A < m$
Lasso Regularisation level	All the levels tried during training
Critical p-value	$cp \in [0.9, 0.001]$

Table 3.1: Hyperparameters suitable for inclusion in a grid search for the MSDA and some suggested values or ranges. n is the number of input variables and m is the number of training examples.

- **Exploration / Exploitation Trade-off** When building a weight, the neurons that are chosen to connect to it are picked either because they have proved useful already (exploitation) or haven't been explored yet (exploration). The trade-off between the two is controlled by a hyperparameter, δ which can be explored as part of a grid search. An alternative approach is to alternate from one iteration to the next between exploring ($\delta = 1$) and exploiting ($\delta = 0$).

- **Schedule for emptying the used weights list** We have found that emptying the used weights list when the algorithm reaches a point where adding new weights has very little effect on validation error is a useful heuristic. If five iterations of adding and removing weights produces no improvement in validation error, that should trigger an emptying of the used weight set.

- **Stopping Criteria** In common with many iterative learning algorithms, the MSDA can be terminated according to a number of criteria including a limit on run time or execution cycles, a target validation error level or a consistent rise in validation error over several iterations.

Some of the hyperparameters can be explored using a grid search. Those that are suitable are listed in table 3.1 with some suggested values over which to perform the search (n is the number of input variables and m is the number of training data points). More work is required on better methods for automatically choosing some of the values such as the number of training epochs and the used weight list emptying schedule.

3.3.11 *Analysis of the Algorithm*

The MSDA is incremental and produces a solution that improves (or at worst gets only a little worse) from iteration to iteration, meaning that it can be stopped at any time and a solution of

some quality will be available. As the model grows and shrinks, the number of parameters to be estimated at each iteration remains a subset of those that might be considered, allowing data sets to be smaller. The algorithm is capable of considering more potential weights than the number of training examples available as long as the number of weights in use at any iteration remains smaller than the training sample size.

Structure complexity is restricted by the shape of the weight order probability distribution, $Po()$, but rather than imposing restrictions a priori, the algorithm attempts to discover the restrictions as it progresses. This works well on functions where certain orders dominate, but is not suited to finding randomly, sparsely distributed weight order patterns. Domain knowledge can be introduced by biasing the order picking distributions towards orders that are known to dominate in a given function or by limiting the highest allowed order. The neuron picking distribution can also be set to favour inputs that are known by a domain expert to be more useful than others.

The algorithm is unlikely to become trapped in a local optimum as the weight picking distributions can always be made to allow new, untried weights to be added. It can, however spend long periods on an error plateau, fruitlessly adding then removing the unhelpful weights, making only small changes to the weight picking distributions. Once weights have been added and removed, they cannot be added for a certain number of iterations, so the algorithm can eventually find some new useful weights to help it move off the plateau.

At each iteration of the algorithm, re-fitting the model has complexity $O(mwp)$ where m is the number of training points, w is the number of weights to update and p is the number of passes through the data made by the SGD learning. The size of the weight space to be searched grows in $O(2^n)$ but the number of weights actually considered by the algorithm depends on how quickly it finds sufficient weights, which depends on the structure of the underlying function. A version of the no free lunch theorem [146] applies as on average across all possible functions no approach can do better than repeatedly picking random structures and learning them in turn. However, the assumption is made that many functions underlying real data from the real world have a structure thatmay be discovered more efficiently. The assumptions that the MSDA makes can be controlled to some extent:

- The distribution of weight orders in the function is assumed not to be uniform. An assumption about the shape of the distribution can be imposed (in this work we always favour lower orders over higher ones) and that assumption can be updated in the light of evidence as the algorithm progresses;
- It might be assumed that if a variable is included in the data that it is important, so preference is given during weight selection to variables that have not yet been used in the model. Alternatively, it might be assumed that a variable that is important at one order is important at others so that well connected variables are favoured.

3.4 Network Dynamics

A dynamic for a MOHN is defined in terms of the way neuron values are updated according to the values of other neurons to which they are connected. The values of X may be set from external stimuli or calculated based on the weights and neuron values of connected neurons. A simple dynamic is achieved by asynchronously updating single neurons one at a time based on the values of their connected neurons and the strengths of those connections. The neuron value X_i is set to 1 or -1 based on connected weights by first calculating the neuron activation a_i

$$a_i = \sum_{j:i\in\mathbf{I}_j} \left(\omega_j \prod_{k\in\mathbf{I}_j\setminus i} X_k \right) \tag{3.34}$$

where $j : i \in \mathbf{I}_j$ makes j iterate over each weight connected to X_i, ω_j is the weight value associated with W_j and $k \in \mathbf{I}_j \setminus i$ iterates over the indices of every neuron connected to W_j, except X_i itself. A neuron's output is then calculated using the threshold function in equation 3.35.

$$X_i = \begin{cases} 1 & \text{if } a_i > 0 \\ -1 & \text{otherwise} \end{cases} \tag{3.35}$$

Setting the values of X to any chosen pattern and then repeatedly applying equations 3.34 and 3.35 to neurons selected uniformly at random without replacement causes the MOHN to move to an attractor state, from which those equations cause no further change to the neuron values (see algorithm 9). The basin of attraction for any attractor is the set of starting patterns that lead to it by this process. As the order in which the neurons are updated is randomised, the same starting point may lead to different attractors in repeated trials [2]. Algorithm 9 describes the algorithm for settling from a pattern to an attractor.

To prove the algorithm is guaranteed to terminate, it is sufficient to show that the neuron updates never lead to an increase in the energy function of equation 3.2. Consider neuron X_k. If the application of equations 3.34 and 3.35 cause no change to the sign of a_k, then X_k and the network energy remain unchanged. If the sign of a_k differs from that of X_k, then the neuron value undergoes a change of sign.

2 For a trivial example of this, consider a network with only two nodes with a single positive connection between them. The starting point $(-1, 1)$ will lead to the attractor state $(1, 1)$ if the first neuron is updated first and to $(-1, -1)$ if the second neuron is updated first.

Algorithm 9 Settling a trained MOHN to an attractor point

```
X ← x                                   ▷ Choose an initial state to settle from
repeat
    ch = FALSE                          ▷ Keep track of whether or not a change has been made
    visited = ∅                         ▷ Keep track of which neurons have been visited
    repeat
        i = rand(i : i ∉ visited)       ▷ Pick an unset neuron uniformly at random
        temp = X_i                      ▷ Make a note of its value for later comparison
        Update(X_i)                     ▷ Update the neuron's output using equations 3.34 and 3.35
        if X_i ≠ temp then
            ch = TRUE
        end if                          ▷ If a change was made to the neuron's output, note the fact
        visited = visited ∪ i           ▷ Add the neuron's index to the visited set
    until |visited| = n                 ▷ Loop until all neurons have been updated
until !ch                               ▷ Loop if any neuron value has changed
```

Let x be the vector of neuron values before the flip and let x' be those values afterwards. The only difference between x and x' is that x_k has its value negated. At this point, a_i and x_k differ in sign. The difference between $U(x)$ and $U(x')$ is in the field of X_k so

$$U(x) - U(x') = -a_k x_k - (-a_k x'_k) \tag{3.36}$$

Noting that $x'_k = -x_k$, this is re-written as

$$U(X) - U(X') = -a_k(x_k + x_k) = -2x_k a_k \tag{3.37}$$

As noted above, the signs of a_k and x_k must differ if a change to the neuron value was made, so the product must be negative, making the difference positive. This proves the energy must fall in response to a change in value of a neuron. Given that there are a finite number of states the network can take, it follows that the neuron updates must ultimately lead to a local minimum in the energy function.

The pattern of activation across the neurons represents the attractor point, which is a local minimum in the energy function. The algorithm updates neuron values one at a time so the minimum is local in the sense that a change in the value of any single neuron will not cause the function's output to change. The energy function may contain plateaux, in which case the network behaviour can be tuned to either move about a plateau (allowing a neuron to flip its value as long as the energy does not increase) or settle on the first pattern it encounters on the plateau (insisting on an energy decrease from a neuron flip).

The traditional application of this type of settling process is the use of a network as a content addressable memory or de-noiser. The next section considers the simple dynamic described above along with others as tools for function optimisation.

3.5 MOHNs and Local Search

This section considers the use of a MOHN as a grey box fitness function model (FFM). Section 2.2 introduces the idea of grey box FFMs and makes the distinction between 'weak' and 'strong' constraints. A MOHN represents the fitness function in an explicit structure of weak constraints among groups of inputs. Minimising the total cost of violated constraints is equivalent to maximising the output of the function and the search for a maximal value may be guided by the structure and value of the constraints.

Any MOHN can be represented as a set of constraints, $\mathbf{C}_j, j = 1 \ldots m$. Each constraint is defined by an index set, $\mathbf{I}_j$, which specifies which inputs are constrained and a weight ω_j whose sign specifies whether the product of the values across the inputs defined in $\mathbf{I}_j$ should be positive or negative and whose magnitude defines the strength (or importance) of the constraint. The $\mathbf{I}_j$ and ω_j values of the constraints equal those in the MOHN function definition, so it follows from the fact that the MOHN function is a basis that this form of constraint definition is universal. That is to say that any fitness function in $f : \{-1, 1\}^n \rightarrow \mathbb{R}$ can be represented as a set of weighted constraints on the sign of the product across unique subsets of inputs. The number of patterns across any $\mathbf{I}_j$ that satisfy the constraint defined by the sign of ω_j is $\binom{n}{|\mathbf{I}_j|}/2$.

Here is an illustrative example. Let $f(X) = 3X_1X_2 - 2X_1 + X_2$. The constraints may be written as

$$
\begin{aligned}
&C_1 : X_1X_2 = 1, \quad \omega = 3 \\
&C_2 : X_1 = -1, \quad \omega = 2 \\
&C_3 : X_2 = 1, \quad \omega = 1
\end{aligned}
$$

They may be read as "The product of X_1 and X_2 should be positive, with an importance of three. X_1 should be negative, with an importance of two and X_2 should be positive, with an importance of one". Constraints C_2 and C_3 are incompatible with C_1 so it is not possible to satisfy all three. In this example, it is easy to see which constraints should be satisfied, guided by the weights. C_1 is most important and can be satisfied at points (1,1) and (-1,-1). Satisfying C_1 leaves a conflict between C_2 and C_3. As C_2 is more important, it should be satisfied, which defines $X_1 = -1$, which dictates the choice of (-1,-1) as the point that satisfies C_1. Alternatively, $f(X)$ has a global maximum at $X = (-1, -1)$ and a local hill climb of $f(X)$ has a local maximum at $X = (1, 1)$. This can be escaped by considering the steps that are acceptable to C_1 as the

search neighbourhood. General algorithms for attempting to use the constraints to guide local search are considered in this section.

Fitness function models have been proposed for problems where evaluating the true fitness function is expensive and a model that can be evaluated more easily can be built from fewer fitness evaluations than are required to find an optimal input [135]. The fitness function model might be used to guide a search in an iterative process of modelling and sampling. This approach is taken by EDAs and Gaussian process optimisation, for example. Alternatively, the fitness function could be modelled exactly and the model searched without further reference to the real fitness function. In this latter case, if the fitness function samples are noise free and the right parameters can be found, a basis function such as a MOHN will be able to reproduce the fitness function with sufficient accuracy to allow the optimal input to be generated from the model alone. All that remains is to search the model. This is the approach that is taken in this thesis.

For a full fitness model to be the most efficient route to an optimal solution requires the combined modelling and model search to be faster (if speed is the measure of efficiency) than a search of the fitness function without a model. Whether this is true depends on the efficiency of the modelling process and the search method employed. Here we compare searching the fitness function with a range of local search methods to modelling a fitness function with a MOHN using MSDA and then searching the MOHN using grey box versions of those same local methods. The local search methods considered are random restart hill climb (RRHC), iterated local search (ILS), variable neighbourhood search (VNS) and simulated annealing (SA).

The connections in a MOHN define independence and parity constraints among inputs, which brings the following two advantages to using a MOHN fitness model over a black box fitness function to implement local search algorithms:

1. Independence among inputs allows the MOHN to support **incremental evaluations** as the effect on the output of a local change to the current input can be evaluated by considering only the weights and neurons connected to the input(s) that have changed;

2. The parity constraints defined by the weights in a MOHN can provide information that can be used to guide a heuristic search.

3.5.0.1 *Incremental Evaluations*

Consider a MOHN with inputs currently set to x as part of a local maximisation search with neighbourhoods of size one. Let X_i be a single variable being considered for a change, which would move the network state from x to x'. The value of X_i should be changed if $f(x') > f(x)$. A black box fitness function would require the full evaluation of $f(x')$. In graphical models such as a MOHN the effect of changing X_i can be calculated from the nodes and weights connected to X_i alone. This provides an improvement in computational complexity.

3.5.0.2 *Weights as Constraints*

Each weight, $W_j = (\omega_j, \mathbf{I}_j)$ in a MOHN represents a weak constraint on the values of the neurons whose indices are contained in the set $\mathbf{I}_j$. The weight has a value, ω_j, which defines two things about the constraint. The sign of ω_j determines the required sign of the product of the values in the connected neurons, indexed in $\mathbf{I}_j$. The magnitude of ω_j determines the relative importance of the constraint. The constraints are applied to the output of the MOHN function (equation 3.1) so that if the parity of the values in the neurons indexed by $\mathbf{I}_j$ agree with the sign of ω_j, then ω_j is added to the output sum that defines the MOHN function. Otherwise, ω_j is subtracted from the output sum. Each neuron may feature in more than one weight and those weights may apply conflicting constraints on that neuron at some points in the input space. In some functions, all of the constraints can be satisfied and in others the conflicting constraints mean that only subsets of the constraints may be satisfied. The output of the MOHN function has a global maximum at the point where the input values satisfy the subset of weight constraints that maximise the difference between the sum of the magnitudes of satisfied constraints and the sum of magnitudes of those that are not satisfied.

Part of this thesis addresses the question of how the explicit representation of constraints as weights can be used to guide a heuristic search. Some functions may be searched efficiently by a process of setting the values across connected subsets of neurons so that they satisfy the constraint defined by the sign on their weight. Other functions have a structure that does not bring such a benefit. Later experiments will demonstrate this further. For example, section 4.14 describes optimising Ising models, which provide no higher order information to guide a search, whereas section 4.15 describes experiments that optimise a k-bit trap function, which can be searched very efficiently using the weights as a guide, once the structure is learned.

There are many ways in which a fitness function model might be searched, but this work concentrates on implementing versions of local search algorithms that can take advantage of the network structure. The next sections present methods for searching a MOHN inspired by random restart hill climb, iterated local search, variable neighbourhood search, simulated annealing and tabu search.

3.5.1 *Random Restart Hill Climb*

Random Restart Hill Climb (RRHC) is a simple way to explore the attractors in a MOHN. Starting points are picked by setting each neuron to 1 or -1 at random after which the network is settled using algorithm 9. The advantage of RRHC is that there is very little computational complexity in each restart, but the disadvantage is that restarts in the same basin of attraction will lead repeatedly to the same attractor point. This may be desirable if the purpose of the

analysis is to characterise the relative sizes of the basins of attraction, but can be quite inefficient if the purpose is to find a global maximum.

Algorithm 10 Random Restart Hill Climb

for a MOHN, $M = (X, \mathbf{W})$
$its = 0$ ▷ Initialise iteration count
repeat
  $X_i = \text{rand}(-1, 1)\ \forall i$ ▷ Initialise to a random starting point
  X = HC(M) ▷ Apply algorithm 9 to settle to a local maximum
  $its = its + 1$
  $S = f(X)$ ▷ Score the found pattern
  Record S and X if best sp far
until $S >= target$ OR $its = max$ ▷ Stop when sufficient patterns or a good enough pattern is found

3.5.2 *Weight Satisfaction Search*

High order weights encode weak constraints among several input variables, offering an insight into candidate moves in a variable sized neighbourhood. Any neurons that are not connected (i.e. there is no path between them in the weights) may be optimised separately and those that are connected will often form smaller subsets of neurons over which it may be possible to find optimal values. The weights in some MOHN structures suggest higher order steps, allowing a variable neighbourhood size to be searched, restricted by combinations defined by their weights. Algorithm 11 describes the process of settling a network by its weights.

The number of patterns tried when finding those that maximise the network output is 2^o where o is the order of the weight so networks with high order weights can produce slow searches. The search can lead to local optima so may need to be repeated. The simplest approach that we propose is a random restart weight satisfaction search, (RRWSS), which repeats algorithm 11 from random starting points.

3.5.3 *Iterated Local Search*

Iterated Local Search [90] replaces the random restart of RRHC with a restart from a position that has something in common with the current attractor point. The move away from the attractor, often referred to as the 'kick' [89] is usually designed based on some knowledge of the problem being solved. In the case of a MOHN, that knowledge is encapsulated in the weight structure so the kick can involve changing the value across subsets of neurons based on their pattern of connectivity. A MOHN ILS kick makes a high order jump by applying the weight

Algorithm 11 High Order Weight Satisfaction Search

X_i =rand$(-1, 1)$ $\forall i$ ▷ Choose a random starting point
repeat
  $ch = FALSE$ ▷ Keep track of whether or not a change has been made
  $visited = \emptyset$ ▷ Keep track of which weights have been visited
  repeat
    $W_j = rand(W_j : j \notin visited)$ ▷ Pick a random unvisited weight
    $temp = \{X_i : i \in \mathbf{I}_j\}$ ▷ Make a note of its connected values for later comparison
    $\{X_i : i \in \mathbf{I}_j\} = \underset{X_i : i \in \mathbf{I}_j}{\text{argmax}}(f(X))$ ▷ Find the pattern across the connected neurons that maximises network function output
    if $X_i : i \in \mathbf{I}_j \neq temp$ **then**
      $ch = TRUE$
    end if ▷ If a change was made to any neuron's output, note the fact
    $visited = visited \cup W$ ▷ Add the weight to the visited set
  until $\|visited\| = n$ ▷ Loop until all weights have been visited
until $ch = FALSE$ ▷ Loop if any neuron value has changed

satisfaction search to a subset of weights in the network. The fewer weights that have their neurons updated, the smaller the effect on the network state of the kick.

Two approaches to the ILS kick in a MOHN are considered. The first uses an exhaustive search over the states that neurons connected to W_j can take, finding the state that maximises the network's output given the current value of all the other nodes, $X_i : i \notin \mathbf{I}_j$. The second updates the neurons connected to W_j according to the sign of ω_j alone. As ω_j is a scalar, the contribution of any state across its connected neurons can only be one of two possible values: ω_j or $-\omega_j$, depending on the sign of $\prod_{i:i\in\mathbf{I}_j} X_i$. The kick does not need a search, it simply chooses a random pattern of values across the members of $\mathbf{I}_j$ so that the sign of their product matches that of ω_j.

3.5.4 *Local Optimum Suppression Search*

Treating each of the local optima as memories in a content addressable memory suggests the idea of removing (or un-learning) local optima as they are found to avoid subsequent searches rediscovering them. We propose this approach and call it local optimum suppression search (LOSS). The attractor removal step involves applying the weighted Hebbian learning rule with a negative learning rate. The forgetting rule is

$$\omega_j = \omega_j + \eta f(x) \prod_{i \in \mathbf{I}_j} X_i \tag{3.38}$$

Algorithm 12 ILS with High Order Kicks

for a MOHN, $M = (X, \mathbf{W})$

repeat

X =HC(M) ▷ Perform a local search using algorithm 9

if $f(X)$ is not satisfactory **then**

Choose $\mathbf{A} \subset \mathbf{W}$ ▷ Choose a random subset of weights

for all $W_j \in \mathbf{A}$ **do**

$\omega_j = \underset{X_i : i \in \mathbf{I}_j}{\text{argmax}}(\hat{f}(X))$ ▷ Find neuron values connected to W_j that maximises $\hat{f}(X)$

end for

end if

until A satisfactory solution is found or timeout

Algorithm 13 ILS with Parity Preserving Kicks

for a MOHN, $M = (X, \mathbf{W})$

repeat

X =HC(M) ▷ Perform a local search using algorithm 9

$its = 0$ ▷ Initialise an iteration count

Choose $A \in W$ ▷ Choose a random subset of weights

for all $W_j \in A$ **do**

$\prod_{i:i \in \mathbf{I}_j} X_i$ =parity(ω_j) ▷ Set the values across the connected neurons to any random combination where the sign of the their product agrees with the weight's value, ω_j

end for

$its = its + 1$

until $S \geq target$ OR $its = max$ ▷ Stop when sufficient patterns or a good enough pattern is found

where x is the value across the inputs that represents the local optimum to remove and $-1 < \eta < 0$ is the learning rate.

Algorithm 14 Local Optima Suppression Search.

for a MOHN, $M = (X, \mathbf{W})$
$\mathbf{W'} = \mathbf{W}$ ▷ Copy the weight values
repeat
  $X_i = \text{rand}(-1, 1)\ \forall i$ ▷ Initialise to a random starting point
  X=HC(M) ▷ Apply algorithm 9
  if $f(X)$ is not acceptable **then**
    Apply learning rule 3.38
  end if
until One of the attractors is of sufficiently high quality or the network becomes too degraded
$\mathbf{W} = \mathbf{W'}$ ▷ Restore the weight values from the copy

Attractor points may be sampled by picking uniformly random starting points and settling the network. Once an attractor has been found, it is evaluated by the true fitness function to verify its score as the output from the MOHN will soon become degraded as weights are changed. If the score is not sufficient, then $X = x$ is deemed to be a local optimum and is learned with a negative learning rate, η. A smaller learning rate leads to a longer search as local optima are suppressed more slowly. In some cases a higher learning rate leads to a solution much faster but in other cases destroys the function to the extent that a solution is never found. A rate of $\eta = 0.3$ was found to speed the search without damaging the desired attractors in most of the experiments carried out for this thesis.

The unlearning can also remove true optima so if an acceptable solution is not found within a small number of iterations, the network structure becomes too degraded and renders the MOHN unable to represent the true maximum. At this point, which can be identified by tracking the agreement between the MOHN's output at local optima and those of the real fitness function, the MOHN's weights must be restored to their original values and the process repeated. This suggests that the LOSS algorithm may be best suited to functions where the basins of attraction for local optima are large, but few. In cases where there are a great many local optima, it is unlikely to be able to remove sufficient of them in time to find the global optimum before the MOHN becomes too degraded.

3.5.5 *Simulated Annealing*

Simulated annealing (SA) [112], [77] attempts to find the maximum of a function by hill climbing where each step has a probability of being taken proportionate to the size of the improvement it will make. This allows down hill steps to be taken in the hope that local maxima can be

avoided. The annealing process is driven by a parameter known as the temperature, T, which controls the influence of the function on the search (or, conversely controls the degree of randomness of the search). The temperature and the stochastic nature of the search combine to attempt to overcome two problems. Firstly, when the temperature is high, the search can make repeated down hill steps and so escape quite large local maximum attractors. Secondly, as the temperature decreases, the search still has the capacity to make smaller downhill steps to escape smaller local maxima. The hope is that the right cooling schedule will allow the algorithm to find the attractor to the global optimum before its temperature is sufficiently low to rule out an escape.

To apply simulated annealing, it is necessary to define a neighbourhood for each state. The neighbourhood of state x, $N(x)$ is the set of states that the algorithm is permitted to visit in a single step from state x. It is also necessary to define a transition probability function, $P(y, y', T)$ that calculates the probability of the algorithm moving from state x to state $x' \in N(x)$, at temperature T where $y = \hat{f}(x)$ and $y' = \hat{f}(x')$.

For a MOHN, there are a number of ways to define $N(x)$. The simplest is to allow any pattern within a Hamming distance of 1. This is the neighbourhood used by the MOHN settling algorithm 9, which updates neurons one at a time. Higher order neighbourhoods can also be defined, similar to those reached by the ILS kicks, by considering sets of connected nodes and setting their values according to the weights that connect them. In this way, SA can be made to act like ILS where the kicks are possible at any point, not just the local optima. The probability transition function proposed by Kirkpatrick [112] is

$$P(y, y', T) = \begin{cases} 1 & \text{if } y' > y \\ exp((y' - y)/T) & \text{otherwise} \end{cases} \tag{3.39}$$

A symmetrical version of equation 3.39 may also be used in which the probability of acceptance climbs towards 1 when $y' > y$ rather than jumping straight to it. This is implemented with equation 3.40, which allows the probability of a change to be proportionate to the size of the change in both directions. Figure 3.2 shows the function $P(y, y', T)$ at various values of T. At high temperature, the function is almost uniform at 0.5 and at low temperature, the probability of taking a negative step is close to zero across almost all of its range.

$$P(y, y', T) = \frac{1}{1 + exp((y' - y)/T)} \tag{3.40}$$

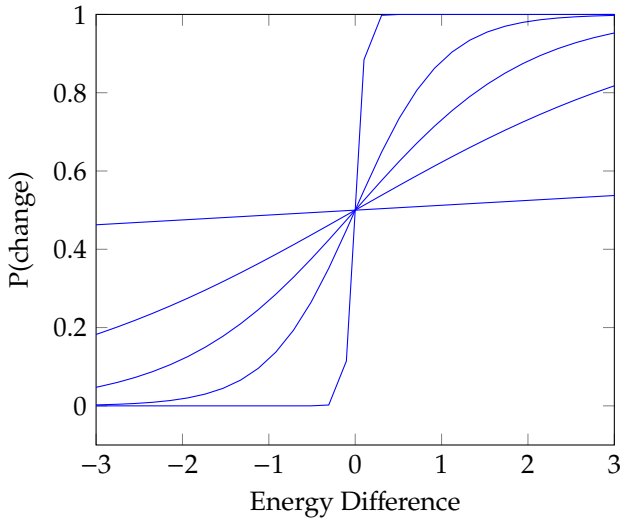

Figure 3.2: Probability of accepting a change by the size of that change at various temperatures during simulated annealing using equation 3.40. T varies from 20 (the flat line) to 1/20 (the step).

Algorithm 15 Simulated Annealing on a MOHN

$T \leftarrow$ high
$x \leftarrow rand(\{-1, 1\}^n)$ ▷ Choose a random starting point
$X^* \leftarrow x$ ▷ X^* will store the best solution so far
$best \leftarrow \hat{f}(x)$ ▷ *best* will store the best score so far
repeat
  Pick $x' \in N(x)$ at random
  Let $y \leftarrow \hat{f}(x)$ and $y' = \hat{f}(x')$
  if $y' >$ *best* **then**
    $X^* \leftarrow x'$
    best $\leftarrow y'$
  end if
  $x \leftarrow x'$ with probability $P(y, y', T)$
  Reduce T
  if The network is stable **then**
    $x \leftarrow rand(\{-1, 1\}^n)$ ▷ New random starting point
    $T \leftarrow$ high
  end if
until A solution is of sufficiently high quality or timeout
X^* contains the optimal pattern found

Algorithm 15 describes simulated annealing on a MOHN. Applying simulated annealing to models like the MOHN provides an efficiency gain over applying SA to a black box function as the decision of whether or not to change each variable's value can be made with reference to

only those other variables that are connected to it rather than requiring a full evaluation of the function output. The experiments with simulated annealing in this thesis used a neighbourhood of Hamming distance 1 and the sigmoid update probability of equation 3.40.

3.5.6 *Choosing a Search Method*

The efficacy of the different search methods depends to a large extent on the structure of the network. If the MOHN may be partitioned and the partitions are small, then optimising the partitions with a high order search may be possible. If the graph is densely connected, particularly at low order, then this approach is less applicable. Examining a visualisation of the MOHN structure may inform this decision. In situations where the structure of the problem is unknown, it is sensible to start with the simpler methods such as RRHC and progress to others should that fail.

LOSS is suitable for functions with a smaller number of local optima with large basins of attraction. RRHC and ILS may fail on such functions as restarts are very likely to put the algorithm within the basin of attraction of one of the local optima. Functions with very many local optima are not suitable for searching with LOSS as it will not be able to remove them all.

3.6 Network Analysis

An advantage of a MOHN over other neural networks, such as the MLP is that the structure of the function is reflected in the weights in a way that is easier to interpret and analyse. The previous section described how that explicit structure could be used to direct a search. This section describes methods for analysing function complexity and comparing the MOHNs in an ensemble as well as presenting a method for visualising the weights in a network.

3.6.1 *Complexity and Regularisation*

Section 2.1.1.2 describes the bias/variance trade-off in which model complexity is controlled to balance between under and over fitting of the training data. This is often controlled by either limiting the number of parameters in a model (using measures such as AIC or BIC for example) or by restricting the size of parameter values (for example the lasso restricts the L_1 norm). Models in which the parameters are learned by an iterative process, such as training an MLP by gradient descent may also make use of early stopping to reduce model variance.

These approaches to regularisation can be employed when training a MOHN. The lasso learning rule described in section 3.2.3.2, for example employs L_1 regularisation to the weight values. The structure discovery algorithm described in section 3.3 regularises by removing

weights and an independent test set may be used along side early stopping of the stochastic gradient descent training described in section 3.2.3.3.

3.6.1.1 *MOHN Ensembles*

One method of addressing variance in models such as neural networks is to build an ensemble, which is a number of different networks, all trained on the same data (or samples from it), and generate the final output by aggregating the individual network outputs. In a MOHN, a given weight encodes a defined contribution that is the same in every model with the same input variables. Given two networks of size n built from measurements from the same variables in the same order in X, any weight connecting a given subvector of X in one network will play the same role as it does in the other. This allows the networks in an ensemble to be compared and near duplicates to be removed. Given two weight sets $\mathbf{V}$ and $\mathbf{W}$, $\mathbf{V} \cup \mathbf{W}$ denotes the structure that results in building a network that contains all the weights in both sets, $\mathbf{V} \cap \mathbf{W}$ denotes the set of weights that appear in both and $\mathbf{V} \setminus \mathbf{W}$ denotes the set of weights in $\mathbf{V}$ that are not in $\mathbf{W}$.

The fact that weights across different MOHNs in an ensemble share the same meaning (in the sense that they encode the interaction between the same set of inputs) means that the members of a MOHN ensemble can be further processed in a way that is not possible with an MLP ensemble. This section describes two methods of combining the MOHNs in an ensemble.

3.6.1.2 *Ensemble Intersection*

Reasoning that any weight that has been chosen by all the MOHNs in an ensemble should be kept, the intersection of all the MOHNs in an ensemble is calculated. The resulting MOHN contains only those weights that are present in every MOHN in the ensemble. This process yields a new structure, but the weight values need to be re-calculated so the training data is used once more to learn new values for the given weights. Finding the intersection of the MOHNs in an ensemble is achieved by starting with, E_1 the first MOHN in the ensemble. For each of the remaining MOHNs in the ensemble, E_k, any weight that is in E_1 but not in E_k is removed from E_1. Any weights remaining in E_1 at the end of this process must be in every MOHN in the ensemble. If no weight is present in every MOHN in the ensemble, the intersection will be empty. How likely it is that this will happen depends on how each MOHN was trained. If the assumptions made by the MSDA are similar for each MOHN, then that algorithm has been found to include a similar weight set across an ensemble of different training sets (picked during cross validation, for example).

3.6.1.3 *Ensemble Average*

A common practice in data mining is to build an ensemble [80] of models (either of the same type of a variety of types) and to take the average output across the ensemble in response to a given input. The ensemble may be built on the same data but across a variety of model

structures, or on different training data subsets (or both). Calculating an ensemble average is inefficient because it requires a large model set and many evaluations, as pointed out by Bucilua et al. [22] who propose a solution that involves training a new single model to mimic the output of the ensemble.

MOHNs have the quality whereby the average of the output of a number of MOHNs is equal to the output of the single MOHN with weights that are an average of those in the ensemble. The average MOHN is calculated by summing the value for each W_j across the networks in the ensemble and dividing the result by the number of networks it contains. This requires a method of matching weights across networks when different networks contain different weights.

Take an ensemble of e MOHNs with weight sets $\mathbf{W}_1 \dots \mathbf{W}_e$ and a shared neuron set, X. $\mathbf{I}_{jk}$ represents the set of connected neuron indices for weight j in $\mathbf{W}_k$ and ω_{jk} represents that weight's value. Let $\mathbf{U} = \cup_{k=1}^{e} \mathbf{W}_k$ be the union of all the weight sets, W_{ju} be weight j in $\mathbf{U}$ and $\mathbf{I}_{ju}$ be the neuron set connected by weight j in $\mathbf{U}$.

Define a mapping, $w(j,k,\mathbf{U})$ that looks in the weight set $\mathbf{W}_k$ for a weight with the same connection set index as weight j from $\mathbf{U}$ and returns the value of the weight in $\mathbf{W}_k$ that has the same connection set as $\mathbf{W}_{ju}$ or zero if it is not present in $\mathbf{W}_k$:

$$w(j,k,\mathbf{U}) = \begin{cases} \omega_{ak} : \mathbf{I}_{ak} = \mathbf{I}_{ju} & \text{if } \mathbf{I}_{ju} \in \mathbf{I}_{jk} \\ 0 & \text{otherwise} \end{cases} \tag{3.41}$$

where a acts as a selector, identifying the weight in $\mathbf{U}$ with the same connectivity set as $\mathbf{W}_{jk}$.

The average output, $\bar{f}(X)$ over all $\mathbf{W}_k, k = 1 \dots e$ is

$$\bar{f}(X) = \frac{1}{e} \sum_{k=1}^{e} \sum_{j=1}^{|\mathbf{W}_k|} w(j,k,\mathbf{U}) \prod_{i \in \mathbf{I}_{jk}} X_i \tag{3.42}$$

The ensemble average weight value, $\bar{\omega}_j$ for each weight in $\mathbf{U}$ is defined as

$$\bar{\omega}_j = \frac{1}{e} \sum_{k=1}^{e} w(j,k,\mathbf{U}) \tag{3.43}$$

Substituting equation 3.43 into equation 3.42 gives

$$\bar{f}(X) = \sum_{j=1}^{|\mathbf{U}|} \bar{\omega}_j \prod_{i \in \mathbf{I}_{ju}} X_i \tag{3.44}$$

proving the equivalence between the average of the ensemble outputs and the output of the average MOHN across the ensemble. By averaging the MOHNs in the ensemble into a single MOHN, the requirement to maintain e different models and make e different predictions each time is reduced to the task of using a single MOHN. The resulting MOHN will be at least

as large as the largest MOHN in the ensemble and has the potential (if none of the ensemble members share a weight) to be the same size as the sum of the ensemble members. However, if the ensemble members share many weights, the average model will not be too much larger than any single member. Unlike the case where the ensemble intersection is calculated, the average MOHN does not need to have its weights re-calculated.

3.6.2 *Visualising Networks*

Simply listing the connection patterns and weights of a MOHN does not make interpretation easy. Methods of visualising different aspects of a MOHN's structure are also needed. Figure 3.1 shows a small network and depicts weights with different shapes connecting the neurons. That is fine as an illustration of the conceptual structure of a MOHN but does not scale well for visualising large networks trained on real data.

The full network structure and an indication of the size of the weights can be represented by arranging the weights in rows in combinatoric[3] order and the neurons in columns. A weight is represented on a single row by plotting a point in the column corresponding to each neuron it connects. The colour of the plotted point indicates the size of the weight. In the examples shown here, red indicates negative weights and green indicates positive. Figure 3.3 shows an example for a small network. Further examples are given later to illustrate other experiments.

3 Combinatoric order starts with patterns with only 1 bit set to 1, followed by those with 2, then 3 and so on. Within each group, the order prioritises weights to the left of the vector. E.g. 100, 010, 001, 110, 101, 011, 111.

Figure 3.3: An example visualisation of the weights of a MOHN.

3.6.3 *Network Summary Visualisation*

Visualising large full networks is not always practical or instructive. During learning it can be more revealing to see summary information such as the number of weights tried and the number of weights kept at each order. A histogram showing weight counts at each order as network structure discovery progresses can show not only the complexity of the current network, but also the progress being made during learning. See page 148 for an experimental example of this.

3.7 Comparison with Existing Work

Chapter 1 described a number of existing approaches to function approximation, dynamic systems and heuristic search. This section compares the MOHN with some of the approaches described in chapter 1 from a theoretical viewpoint. Chapter 4 presents experiments that compare the MOHN with other approaches on real data.

3.7.1 *Function Learning*

A MOHN is a linear parameter model which, in the case of a full weight set, forms a basis set. As such it displays all of the characteristics of a linear parameter model and may be extended to the set of generalised linear models via a link function. This set includes approaches such as logistic regression and Markov random fields.

Model bias in a MOHN can be controlled with the choice of which weights to include. A fully connected MOHN or a MOHN with only the required weights has no model bias. Model bias can be introduced by excluding weights that are required. This may happen due to the structure discovery algorithm failing to find the right weights, or due to design decisions. Estimation bias may be introduced by the choice of learning algorithm. OLS is unbiased, the lasso introduces estimation bias controlled by the shrinkage parameter, λ and SGD regularises with early stopping, which offers less precise control than the lasso.

The obvious difference between MOHNs and MLPs (including deep networks) is that MOHNs are very shallow. There are no hidden units, which means that interpretation is easier, but the features (i.e. the choice of which weights to include) must be discovered explicitly. Once the features have been chosen the cost function may be minimised in a number of different ways, and there are convex cost functions available for doing this. MLPs need to be trained multiple times to try and avoid local minima, making the task of finding the right model more difficult.

3.7.2 *Structure Discovery and Feature Detection*

Section 2.4 reviewed a number of methods for discovering structure in graphical models and performing the very similar task of selecting features for a regression model. This section considers the MSDA in the context of these methods. Section 2.4 considered methods for structure discovery for graphs such as Bayesian networks and MLPs and for hypergraphs such as Markov random fields and hypernetwork classifiers. It concluded that the latter group are more relevant to this work so this section concentrates on comparing the MSDA with the following existing work: Evolving hypernetworks [75], the clique finding method used

by sDEUM [94], and the greedy L1 method based on grafting [87]. Algorithm 1 outlined a framework for structure discovery algorithms and this section shows how the MSDA fits into that framework.

Algorithm 1 maintains a candidate set **C** of hyperedges (weights in a MOHN). In a MOHN, membership of that set is probabilistic. The MSDA is not presented in a Bayesian framework, though it shares some features of such an approach. Prior knowledge or assumptions about which weights should be included in a model can be expressed in the form of distributions, which are updated in the light of new evidence. The updates are not made according to Bayes' rule, however.

The MSDA takes inspiration from hypernetwork classifiers as it maintains a probability distribution over weight orders, but extends that idea in two ways. The weight order distribution is not initialised to be uniform. Rather it is shaped to favour low order weights in early iterations of the algorithm and allows the mode weight order to climb as lower orders are exhausted. It also extends the idea of maintaining a distribution to the task of picking the nodes that are connected by each weight.

The MSDA also takes inspiration from greedy L1 methods, using a regularised cost function to guide the removal of weights at each iteration. However, rather than use a greedy approach, new weights are picked purely based on the current probability distributions over the weight orders and the nodes. This has the advantage of not having to consider every member of the candidate set, **C** at each iteration. By allowing membership of **C** to be probabilistic and allowing its distribution to evolve the MSDA has the potential to expose the algorithm to a larger set of candidate weights than an L1 method with a fixed candidate set.

Section 4.15 describes a series of experiments in which a MOHN learns the structure and weights required to reproduce a k-bit trap function. Table 4.14 shows that a MOHN was able to learn the correct structure for a 25 bit problem with $k = 5$ in 8 seconds using 1000 data samples. The MSDA converged in an average of 35 iterations, starting with 200 weights and adding 100 at each iteration. This led to around 8000 candidate weights being considered in total. If the grafting algorithm is used and the candidate set is limited to all weight orders under 6, the size of **C** at the beginning is $\sum_{j=1}^{5} \binom{20}{5} = 68,405$. The greedy approach considers more weights on its first iteration than the MSDA does over its entire run. Of course, the k-bit trap problem is well suited to the MSDA as its structure leads the algorithm quickly to the solution.

The MSDA also takes some inspiration from DEUM, in which higher order weights are connected to nodes that form maximal cliques of second order connections. As the probability of a node being joined to a weight is influenced by the values of other weights already connected to it at other orders, there is a pressure towards well connected input subsets (though not, necessarily, cliques) gaining further connections. For that reason, MSDA will be effective on a very similar set of problems to which the DEUM clique finding approach will be effective. Those are problems where inputs that are connected at lower orders are also connected at

higher orders. The iterative approach of adding and removing weights that MSDA employs was preferred over the single growth step (clique finding) followed by an L1 pruning step because it allows the number of weights in the network at any given time to be kept lower, meaning that smaller training sets may be supported. Section 4.14 provides some experimental results showing that a MOHN trained on data randomly sampled from a noise free function is able to reproduce that function perfectly using far fewer samples than DEUM or sDEUM require.

The MSDA adds weights in batches, rather than using the one at a time approach of grafting. The number of weights added may be chosen by the user, creating a hyper parameter to optimise. In cases where the function being learned is noise free (a fitness function model, for example) the number of weights added can be fixed so that the number of weights in the model always equals the size of the training set after weight addition but before weight removal. Table 3.2 provides a summarised comparison of MSDA with sDEUM, greedy L1 and evolving hypernetworks.

3.7.2.1 *Applicable Scope*

The MSDA relies on the probability distributions over the weight orders and the input nodes to guide its search for candidate weights. If the structure of the target function contains no bias in the weight orders or input variable roles, then there is no information to guide the MSDA. Functions where weights are very sparse and distributed uniformly across the weight orders with no preference for any input over any other will be very difficult for the MSDA to learn. Section 4.1.1.7 describes a set of experiments where a MOHN attempts to learn functions that contain weights of orders that are picked uniformly at random and connected to inputs that are also picked uniformly at random. The MOHNs fail to learn the functions, as expected, but when the same functions are used to generate data to train a multilayer perceptron, the error rates obtained are almost the same. The assumptions made when using MSDA are described in section 3.3.11.

There is no function in $f : \{-1, 1\}^n \rightarrow \mathbb{R}$ that cannot be represented by a MOHN but in practice, there are several sources of limitation that may prevent a MOHN being built to represent a given function. Limitations are due to computing resources (time and memory), data availability and the ability of the MSDA algorithm to exploit regularities in the structure of the MOHN representation of a function. Assume that the structure of a MOHN required to represent a given function is unknown. If the function requires more weights than the number of available training data points (and no more data can be collected), then the MOHN structure and parameters cannot be learned. If data can be collected (or generated) in arbitrary quantity and from arbitrary locations in the data space, then computing resources are the only constraint as, in principle, every possible model can be tried. For even modest numbers of inputs (over 20, say), computing resources start to become the limiting factor, so the search

space must be limited. How effectively the MSDA is able to limit the search space depends on the appropriateness of the assumptions that it makes about the function to be learned, as discussed in section 3.3.11.

3.7.3 *Dynamic Systems*

A MOHN that is fully connected at second order only is equivalent to a standard Hopfield network. Adding higher order connections makes a MOHN equivalent to a high order Hopfield network. As already mentioned, using an exponential link function allows the MOHN to function as a MRF. This work does not address the use of MOHNs to represent probability distributions, but the structure discovery algorithm has the potential to offer a useful method for discovering MRF structure. This will be the topic of future work.

3.7.4 *Heuristic Search*

The form of a MOHN allows the development of search heuristics designed to take advantage of its structure. This allows black box problems (where the function may be sampled, but not examined and where there are no structural clues to guide a search) into grey box problems (where the structure of the function model may be used to guide a search). If the correct model can be discovered, then a set of model searching heuristics can be applied to black box problems that would otherwise require meta-heuristics. For example iterated local search [90] can choose the kicks based on the pattern of connectivity. Smaller efficiencies are also made possible for local searches as each variable can be updated according to the values of the variables it connects to, rather than the full set. There is a cost in building the model originally, but some conditions make that cost justifiable. For example, if the fitness function is expensive to evaluate or more than one potential solution is sought from the same fitness function, searching a model can be more efficient.

Of the search methods reviewed in chapter 1, the MOHN has most in common with DEUM in terms of structure. DEUM is nominally an EDA but is reported in the literature as taking a single generation to build a full model, which is then sampled for optimal solutions, making the approach more like that of fitness function modelling [117] and [94]. This approach is obviously wasteful (though no doubt a step towards proving the capabilities of DEUM) because the selection process required to chose the fitter solutions from a population leads to many samples being evaluated and discarded. Section 4.14 presents some experimental results that demonstrate this claim. Section 3.7.2 has already discussed the differences between DEUM's clique finding approach to structure discovery and that used in the MSDA.

The MOHN can also implement grey box versions of local search algorithms. Examples based on RRHC, ILS, SA, and VNS are proposed in this chapter and the next chapter presents some

experimental examples of their use. The crossover methods proposed by Tintos et al. [137] may also be applicable to searching a MOHN, but exploring that further is left for future work.

3.8 Summary

This section introduced the mixed order hyper network and described methods for estimating parameters, discovering structure and searching models for input points that maximise the function output. Page 4 claims a number of properties for a MOHN, which were demonstrated in this chapter. They are:

1. **Basis Function**: The equivalence to a Walsh basis up to a change in sign for weights whose order has an odd value demonstrates that a MOHN forms a basis in $f : \{-1, 1\}^n \rightarrow \mathbb{R}$. This proof was given in section 3.2.2;

2. **Sparsity**: The MOHN structure discovery algorithm (algorithm 7) was introduced. It attempts to find the non-zero weights of a sparse MOHN in a way that restricts the number of weights in the MOHN and, consequently, the number of data points needed during training.

3. **Linear Parameter Models**: Equation 3.1 represents the MOHN function as a sum that is linear in its parameters.

A number of local search algorithms for optimising the output of a trained MOHN were proposed. They make use of the structure of the MOHN to speed up the search. Two types of efficiency are discussed. When a local change to the inputs is proposed, the effect that it has on the output may be calculated based only on the weights connected to the neurons being changed. This is more efficient than re-evaluating the whole function. The weights of some functions may be used to guide a local search as they explicitly represent the constraints to be satisfied.

A full treatment of all of the claims made at the start of this thesis is reserved until the end of the work. This is presented on page 201.

Initial Hyperedge Orders	
MSDA	Discrete Laplace Distribution by default, others may be user defined if prior knowledge is available
sDEUM	All second order connections
Greedy L1	User defined subset of all possible hyperedges
Evolving HNs	Uniform distribution over all orders
Number of Hyperedges Added per Iteration	
MSDA	User defined or maintain model size less than sample size
sDEUM	Defined by maximal cliques
Greedy L1	One at a time
Evolving HNs	Depends on size of training data set
Hyperedge Removal	
MSDA	L1 regularised parameters that go to zero or a t-test with a hyper-parameter to control the critical p-value
sDEUM	L1 regularised parameters that go to zero
Greedy L1	L1 regularised parameters that go to zero
Evolving HNs	Based on the number of correct classifications an edge is involved in
Iterations	
MSDA	Multiple iterations
sDEUM	Two stages - second order interaction discovery and maximal clique connecting
Greedy L1	Multiple iterations
Evolving HNs	Multiple iterations
Updating of Hyperedge Candidate Set	
MSDA	Order distribution and node distribution evolve with the model
sDEUM	A single switch from second order connections to clique filling
Greedy L1	Greedy search over all candidate hyperedges at each iteration
Evolving HNs	Hyperedge order probabilities reflect the frequency of occurrence in the current model

Table 3.2: Comparing MSDA with sDEUM, greedy L1 and evolving hypernetworks.

4

EXPERIMENTS AND ANALYSIS

4.1 Introduction

This chapter presents the results of a number of experiments that investigate and demonstrate MOHNs in action. It provides experimental evidence for the claims made about MOHNs in chapter 3. The first part of the chapter presents a series of small scale experiments on artificial data. Two subsequent parts follow, the first compares a MOHN to an MLP using a case study on real data describing customer profiles and the second compares the MOHN to some EDAs from the literature on a number of heuristic search problems. Some of the results reported below give execution time in seconds. All of the experiments were run on a PC with a single core 3.4GHz CPU and 16GB of memory. Programs were written in Java.

4.1.1 *Functions and Datasets*

A number of different functions will be used to test the MOHNs. Most will be implemented in Java and sampled to provide training data. This has the advantage that the correct structure of the function is known and can be compared with that of any resulting MOHN. Real data from Experian's *Enhance* data set will also be used.

4.1.1.1 *Symmetry Function*

This function defines the degree of symmetry about the vertical axis of a square image of n pixels where each pixel is a variable in X. The variables in the input are arranged on a $\sqrt{n} \times \sqrt{n}$ grid to form a square black and white image. The fitness score for a pattern is

$$f(X) = \frac{2s}{n} \tag{4.1}$$

where n is the number of variables and s is the number of symmetrically placed variable pairs that share the same value. The correct structure for such a function is a sparsely connected network with second order weights only.

4.1.1.2 *Concatenated XOR*

The XOR function has long been of interest in the development of neural networks as it is not linearly separable. Even for an MLP, XOR functions are interesting because the cost function

had long been considered to possess local minima [15] or at least large plateaux from which gradient descent could not escape [49]. A concatenated XOR function pairs inputs so that each X_i where i is even is paired with X_{i+1} to form an XOR function. The function output is the normalised sum of the XOR of the pairs

$$f(X) = \frac{1}{n/2} \sum_{i=1}^{n/2} \oplus(X_{2i-1}, X_{2i}) \tag{4.2}$$

n is constrained to being even.

4.1.1.3 *Multiple Pyramid Functions*

In these functions, a varying number of target patterns are set as attractor states (local maxima) by building a function based on the Hamming similarity to the closest target. Target patterns can all have the same score, or they can each have different scores (all of which are sufficiently high to ensure they stay locally maximal). Let $\mathbf{T}$ be the set of target patterns with $p = |\mathbf{T}|$ members: $\mathbf{T} = \{T_1, \ldots, T_p\}$.

The fitness function is defined with respect to the Hamming similarity between X and each target pattern $T_j \in \mathbf{T}$.

Let the Hamming similarity between target t_j and pattern X be

$$H(X, T_j) = \sum_{i=1}^{n} \delta_{X_i, T_{ji}} \tag{4.3}$$

where T_{ji} is element i of target j and $\delta_{X_i, T_{ji}}$ is the Kronecker delta function, which is 1 if $T_{ji} = X_j$ and zero otherwise. The function output is the maximal score of all the members of the target set.

$$f(X) = \max_{T_j \in \mathbf{T}}(H(X, T_j)) \tag{4.4}$$

The function is linear in the Hamming similarity to the nearest target pattern, creating a landscape of pyramids. The advantage of such functions is that it is possible to control the number of turning points in a function, and so control one aspect of its complexity. By placing the turning points at random, many different functions can be generated for repeatedly testing a MOHN.

Variable height pyramid functions can be defined by assigning a scaling parameter to the height of each peak. Let $0 < h_j < 1$ be the output of the function when the input is T_j, so the weighted Hamming similarity of a pattern, X to target point T becomes

$$H(X, T_j) = h \sum_{i=1}^{n} \delta_{X_i, T_{ji}} \tag{4.5}$$

Weighting different patterns allows local optima to be placed in the function.

4.1.1.4 *K-Bit Trap Functions*

K-bit trap functions are counting functions based on subvectors of k variables from the input vector. A pattern is split into non-overlapping subvectors, $C \subset X$ of size k and each subvector is scored separately. The function output is the sum of the subvector scores. Each subvector is scored by counting the number of bits set to 1 and letting patterns with k 1s (all of them) score k, but letting patterns with $< k$ 1s score $k - 1 - b$ where b is the number of bits set to 1:

$$f(C) = \begin{cases} k, & \text{if } b = k \\ k - 1 - b, & \text{otherwise} \end{cases} \tag{4.6}$$

and the sum is calculated as

$$f(X) = \sum_{C \subset X} f(C) \tag{4.7}$$

The correct network structure for a k-bit trap function includes first order connections to every neuron, and connections up to order k among the neurons in each subvector, C. Trap functions are deceptive because hill climbing steps of any order less than k increase the score but move away from the global optimum. They are also of interest because the subvectors, C should not be broken from one generation of a GA to the next.

4.1.1.5 *Ising Spin Glass Models*

A spin glass represents a disordered and frustrated magnetic system, which can be represented by a graph in which the nodes, $X = X_1 \dots X_n$ are magnetic *spins* that can be in one of two states: up or down, and the edges represent interactions between the spins. Ising spin glass models have interactions that form a lattice structure, with each spin interacting with its closest neighbours. In a 2D Ising model, each spin interacts with its closest four neighbours on a toroidal plane, meaning nodes at the left and right edges connect to each other, as do those at the top and bottom. Ising models can be defined in higher dimensions, for example a 3D Ising model organises the nodes in a cube so that each node has 6 neighbours. Figure 4.1 shows the structure of a 2D Ising model.

The state of an Ising model is defined by the values of the spins where up=1 and down=-1. Each state has an associated energy, calculated with the Hamiltonian of equation 4.8.

$$H(X) = -\sum_{\langle i,j \rangle} J_{i,j} X_i X_j \tag{4.8}$$

where the sum is over the set of edges in the graph and $\langle i, j \rangle$ indicates that spins i and j have an interaction. When the spin of two connected sites agree with the sign of their connection,

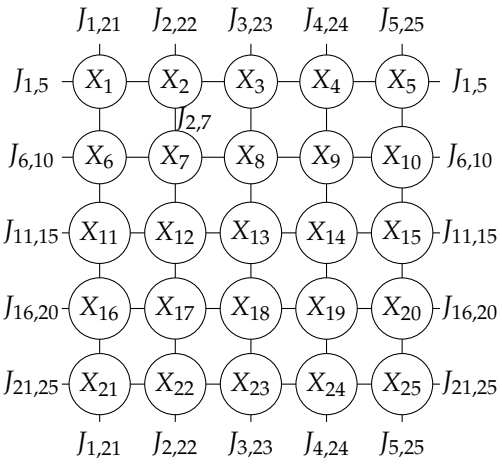

Figure 4.1: A 5×5 Ising model with the toroidal interactions and a single example interaction, $J_{2,7}$ shown. All other interactions, $J_{i,j}$ connect each X_i with X_j where a connection is shown. Note that $J_{i,j} = J_{j,i}$ and is only included as a single edge.

they are said to be aligned. The state in which $H(X)$ is minimised is that in which there is the most alignment between pairs of spins. Depending on the configuration of the connections, an Ising model will have 2 or more global optima and zero or more additional local optima with outputs that are lower than the global maxima.

4.1.1.6 *Graph Colouring Function*

The graph colouring problem involves searching for a way to colour the nodes of a graph so that no two connected nodes share a colour, using a limited palette of colours. The input is encoded in d groups of k bits where k is the number of colours available on the palette and d is the number of nodes in the graph. The colour is encoded by allocating each of the k bits in each block a colour and using patterns where only one bit (that corresponding to the chosen colour) is set to one. The fitness function has two components. One ensures that only one colour is chosen in each group of k and the other counts the number of edges that join same coloured nodes. The function is implemented as follows:

$$f(X) = \frac{|e_d|}{|e_t|} \frac{\sum_{i=1}^{d} \frac{k-|i_1|}{k-1}}{d} \tag{4.9}$$

where $|e_d|$ is the number of edges with a different colour at each end, $|e_t|$ is the number of edges in the graph and $|i_1|$ is the number of inputs in block i with a value 1. The output of the function is 1 when a correct colouring for the graph is present and each block has only one bit set to one. The function has interactions within each block at orders up to k, which control the only-one-colour constraint and additional high order weights between blocks that are connected in the graph.

4.1.1.7 *Randomly Structured Functions*

A fully connected MOHN forms a basis for functions in $f : \{-1, 1\}^n \rightarrow \mathbb{R}$ so a MOHN can be used to generate any function in that domain. Each weight in a MOHN is a tuple, $W_j = (w_j, \mathbf{I}_j)$ where w_j is the value of the weight coefficient and $\mathbf{I}_j$ identifies a set of connected input variables. By choosing a number of weights such that each $w_j \in \mathbb{R}$ is chosen from a uniform random distribution and each $\mathbf{I}_j$ contains a unique subset of input indexes, each chosen uniformly at random from $\{1, 2, \ldots, n\}$ without replacement, a function of random structure can be produced. Aspects of the complexity of the function and of how challenging it might be for an algorithm to discover its structure can be controlled by restricting the order of the weights (i.e. the size of each set, $\mathbf{I}_j$) and the number of weights added.

4.2 Experimental Results

The following sections describe a set of experiments that explore and demonstrate the ability of a MOHN to learn and optimise a number of functions.

4.2.1 *Fully Connected MOHNs*

Section 3.2.2 shows that a MOHN is a universal function model over $f : \{-1, 1\}^n \rightarrow \mathbb{R}$ as a fully connected MOHN provides a basis function in that space equivalent to a Walsh basis. This is claim number 1 made for MOHNs on page 4. For illustrative purposes, the following experiment shows the results of training a fully connected MOHN with the weighted Hebbian learning rule of equation 3.4 on an exhaustive sample of the (input, output) space of a k-bit trap problem over 8 bits where k=4. A Walsh decomposition of the function reveals that the correct structure for the network should contain exactly 32 weights, as shown in table 4.2. Note that the weights and Walsh coefficients agree in value and sign according to the definition in equation 3.6.

4.2.1.1 *Network and Sample Size*

Fully connected networks and full samples are a special case. In reality, most functions worth modelling will have too many inputs to allow a full model or an exhaustive sample. Table 4.1 gives an indication of the size, memory requirement and time to process a full network of varying size. As the table is for illustration only, some simple assumptions will suffice. They are that each weight requires 32 bits to represent its value and n bits to represent its pattern of connectivity. Processing time is linear in the number of weights and experiments show that in

one second, around 35,000 weights can be processed, so this is the figure used to illustrate time in the table.

Neurons	Weights	Bytes	Time	Memory
10	1,024	1,284	3 ms	1K
15	32,768	61,444	1 seconds	62K
20	1,048,576	2,621,444	30 seconds	2M
25	33,554,432	104,857,604	16 minutes	100M
30	1,073,741,824	4,026,531,844	8 hours	4G
35	34,359,738,368	150,323,855,364	11 days	150G

Table 4.1: An indication of the speed at which time and memory requirements grow for training fully connected MOHNs.

It is clear from table 4.1 that for even modestly sized networks, a sparse distribution of weights is required if models are to be built in reasonable timescales. The next section presents experiments with such networks.

Weight	Value	Order	Index	Walsh Coefficient
00000000	0.328	0	0	0.328
00000001	-0.023	1	1	0.023
00000010	-0.023	1	2	0.023
00000100	-0.023	1	4	0.023
00001000	-0.023	1	8	0.023
00010000	-0.023	1	16	0.023
00100000	-0.023	1	32	0.023
01000000	-0.023	1	64	0.023
10000000	-0.023	1	128	0.023
00000011	0.039	2	3	0.039
00000101	0.039	2	5	0.039
00000110	0.039	2	6	0.039
00001001	0.039	2	9	0.039
00001010	0.039	2	10	0.039
00001100	0.039	2	12	0.039
00110000	0.039	2	48	0.039
01010000	0.039	2	80	0.039
01100000	0.039	2	96	0.039
10010000	0.039	2	144	0.039
10100000	0.039	2	160	0.039
11000000	0.039	2	192	0.039
00000111	0.039	3	7	-0.039
00001011	0.039	3	11	-0.039
00001101	0.039	3	13	-0.039
00001110	0.039	3	14	-0.039
01110000	0.039	3	112	-0.039
10110000	0.039	3	176	-0.039
11010000	0.039	3	208	-0.039
11100000	0.039	3	224	-0.039
00001111	0.039	4	15	0.039
11110000	0.039	4	240	0.039

Table 4.2: The Walsh decomposition and non-zero weights of a fully connected MOHN trained on a full sample from the function space.

4.3 Sparse Networks and Sparse Samples

The rest of this thesis investigates the training of sparsely connected MOHNs on data samples of limited size.

4.3.1 *Comparing with a Multilayer Perceptron*

This thesis makes the following claims about the advantages of the structure of connectivity of the weights in a MOHN, compared to the black box model of an MLP

1. Being a linear parameter model, there are no local minima in the squared error cost function when training a MOHN;
2. The MOHN structure lends itself to human interpretation more readily than that of an MLP, both in terms of the role of the inputs and the complexity of the model;
3. The MOHN structure allows heuristic search decisions at variable sizes of local neighbourhood in a way that is more difficult with an MLP;
4. Multiple MOHNs in an ensemble can be structurally combined into a single MOHN whose function is equal to taking the average of the output of each MOHN across the ensemble as a whole.

4.3.1.1 *Local Minima in the Cost Function*

Training a MOHN based on a squared error cost function means that there are no local minima in the error function and the unbiased weight estimate (via OLS, for example) is unique for a given training data set. The cost function when training an MLP using gradient based methods is known to have local minima. Swamy et al. [40] state "Conventional first-order and second-order gradient based methods cannot avoid local minima.". The presence of local minima in an MLP when training using the same cost function can add a source of variance among alternate MLPs trained on the same data. Section 4.15.1 describes a set of experiments in which an MLP is used to learn to reproduce the functionality of a 4-bit trap function over 40 inputs. In those experiments, optimising the hyperparameters of the MLP is made more difficult by the fact that any single hyperparameter set produced variance in the validation error across multiple training attempts. No such variance was found for the MOHN, which minimised the validation error on every trial.

4.3.1.2 *Human Interpretation*

A commonly cited disadvantage of the MLP is the difficulty with which its functional shape can be interpreted by the human user. For example, Jivani et al. [71] recently began a review paper

of neural network rule extraction with "Although neural networks have performed very well for many application domains, one of its main drawbacks is the inherent black-box nature". It can be important to understand the structure of the function implemented by a neural network, for example in certain financial applications where automated decision making needs to be supported by an ability to provide reasons behind a decision.

In addition, we claim that the *complexity* of the function implemented by any particular neural network is not easy to understand from its structure and weight values alone. For example, by performing a full Walsh decomposition of the function represented by an MLP at each epoch during training, we have found that the complexity (in terms of the number of non-zero Walsh coefficients) varies greatly between the first and final training epoch in an MLP of fixed structure [129]. This work also showed that local minima when learning parity based functions coincide with a failure of the MLP to encode higher order interactions among inputs.

By contrast, the interactions among inputs in a MOHN are represented explicitly allowing visualisations of function structure than can (for some functions more than others) be very revealing to the human observer. In addition, the number of weights in a MOHN and the number of inputs each one connects can be interpreted directly as a measure of complexity.

4.3.1.3 *Use as Fitness Function Models*

MLPs have been used very successfully as fitness function models. By following the derivatives the networks make available, many gradient based optimisation methods may be applied. In the case of binary optimisation, we claim that a MOHN makes information explicitly available that can be used for optimisation methods that make use of variable sized search neighbourhoods. These include variable neighbourhood search (VNS) and methods such as iterative local search (ILS) that depend on a perturbation within a defined size of neighbourhood. VNS in a MOHN can (if the fitness function is suitable) be guided very efficiently to input combinations that should be searched. Section 4.15 presents an experiment that demonstrates this claim.

4.3.1.4 *Combining Model Ensembles*

Building an ensemble of models and making a prediction by taking the average output of them all in response to the same input is a well used method for improving model performance and avoiding over fitting. However, the practical disadvantage is that rather than making a single prediction, many are required. When ensembles are large, this can be inefficient. One solution to this problem, proposed by Bucilua et al. [22] is known as model compression and involves training a new single model based on the averaged ensemble output for the training data. Bucilua et al. state that "Often the best performing supervised learning models are ensembles of hundreds or thousands of base-level classifiers. Unfortunately, the space required to store this many classifiers, and the time required to execute them at run-time, prohibits their use

in applications where test sets are large(e.g. Google), where storage space is at a premium (e.g.PDAs), and where computational power is limited (e.g. hearing aids)."

Section 3.6.1.3 shows how an ensemble of MOHNs can be combined into a model whose output is the same as the average across the ensemble for all input patterns without the need for re-training. Section 4.10.2 provides an illustrative experimental example of the process.

4.3.2 *Experiments*

This section experimentally compares MOHNs with MLPs in terms of learning speed, the variance of solutions learned, the ability to avoid local error minima and the ease with which the model may be used as a surrogate fitness function. First, let us compare the complexity of making a prediction and updating the weights of a MOHN with the complexity of doing so with an MLP. In both cases, making a prediction involves each weight once. Weight values are multiplied by connected neuron values and summed. This means the time taken to make a prediction should grow linearly with the number of weights in both cases. Similarly, a single SGD weight update requires time that grows linearly with the number of weights in a MOHN and an MLP as each weight is updated once in a single SGD step. Differences in learning speed, if any are found, can be attributed to the differences in the shapes of the cost function for an MLP and a MOHN.

4.3.3 *Experimental Setup*

Each of the following experiments had the same experimental setup. The first step in MLP training involved a grid search over a defined hyperparameter space. Each combination of hyperparameters was used to train a single MLP on the same training data and validated on the same validation set. The hyperparameter set associated with the network with the lowest test error was selected. In some experiments, several MLPs were trained for each point in the grid search to account for high variability among results gained from identical sets of hyperparameter settings.

Once a set of parameter settings were selected, they were used in every instance of a number of MLPs, each trained on a different sample of data. Each of the experiments described below uses data generated by a known function and each MLP is trained on a small sample where the inputs are picked uniformly at random and the associated outputs are the result of evaluating the chosen input pattern with the known function. Validation data was generated in the same way, with repetitions of training data being avoided in the validation set by comparison with a record of the training points. In some experiments, where it is important to limit the size of the training data, the training and validation sets were generated before each network was trained and no new samples were generated during training. In those cases, the same training data,

presented in the same order, was used for the MLP and the MOHN being compared. Training data inputs were generated by setting each variable from a uniform binomial distribution over $\{-1, 1\}$. Values were not normalised, but the mean of this distribution is zero and its variance is 1.

The parameters that were varied in each test were chosen from the following:

- The learning rate, η
- The rate of decay of the learning rate, τ ($\eta \leftarrow \tau\eta$ on each update)
- How often (every e epochs) the learning rate is decayed
- The momentum rate, α
- The activation of the hidden units, from {Tanh, Logistic, ReLU}
- The number of hidden units in the network
- The number of layers in the network (with an equal number of units in each)
- The range from which the initial random weight values are sampled, given as a single fraction, f and used to set the range of weights from $-r$ to r where $r = l\sqrt{6/(fanin + fanout)}$ where $l = 4$ when the hidden unit activation is logistic and $l = 1$ otherwise. These figures are taken from [13].
- The size of mini batches (1 means SGD is used).
- The dropout probability rate.

For training a fixed structure MOHN, there are fewer parameters to explore. The SGD algorithm has a learning rate that needs to be set and the lasso has the regularisation factor, λ. For learning a fixed network with OLS, there are no parameters to set. When running the MSDA, these experiments restricted themselves to a grid search over values for the following parameters:

- Initial number of weights to start the network with
- Number of weights to add at each iteration of the algorithm
- For SGD:
  - Learning rate
  - Initial p-value for removing weights
  - P-value decay rate
  - Minimum p-value
  - Number of epochs to perform gradient descent on each iteration

- For lasso:
  - Regularisation parameter, λ

During lasso learning, m different values for λ are used to produce m different models, $M_j, j \in [1, \ldots, m]$ each with a level of regularisation determined by λ_j. The actual value of λ depends on the residual correlations between features and the output and is calculated by the lasso learning algorithm. In experiments where data is generated by sampling functions, the choice of λ made by the grid search process is defined by its index, j, not its value. This means that when the results of the grid search are applied to new data, the exact value of λ is re-calculated for that data. This allows a single grid search step to be followed by many repeated experiments on stochastically generated data sets.

4.3.4 *Training Speed, Variance and Local Minima*

With a fixed structure MOHN, the squared error cost function with respect to the network weights is convex and so contains no local minima. The same cost function with respect to the weights in an MLP can contain local minima. Additionally, if the fixed MOHN contains at least the required weights required to represent the function underlying a data set, then it will be able to learn that function from a noise free sample with size equal to the number of weights in the MOHN. This takes place without the need for the learning algorithm to perform the type of feature discovery that takes place in the hidden layers of an MLP. For all these reasons, one should expect a fixed structure MOHN with sufficient weights to learn a function more quickly, with less error variance across trials and from a smaller data sample than an MLP.

4.3.4.1 *Experimental Setup*

In this experiment, a concatenated XOR function as described in equation 4.2 with 20 inputs was used to generate the training and validation data. The sets of hyperparameter values explored in the initial grid search for an MLP were:

Hyperparameter	Grid Set
η Learning rate	{0.3,0.2,0.1,0.05}
α Momentum	{0.8,0.5,0.1,0.05,0.01,0}
τ Learning rate decay	{10,20,50}
Hidden activation	{Tanh, Logistic, ReLU}
Number of hidden units	{10,20,30,50}
Number of hidden layers	{1,2,3}
Random weight range	{0.05,0.01,0.1,1}
Mini batch size	{1,5,10,40,50,200}

The output unit always had a linear activation function and the output data was scaled to fall into the range from zero to one. Each network was trained for 200 epochs on a data set containing 1000 randomly generated examples and the average squared validation error was reported for each epoch.

A grid search revealed the following results. The 30 networks with the lowest validation error all had 1 hidden layer of neurons with tanh activation functions, a learning rate of $\eta = 0.3$ or $\eta = 0.2$ and little or no learning rate decay. Networks with 20, 30 and 50 hidden units were all represented in the best 30, so 20 was selected due to a preference for fewer weights where possible. Momentum and initial weight range were uncorrelated with error. The network hyper-parameter settings for the next step of the experiment were chosen to be:

Hyperparameter	Value
Number of trials	50
η	0.3
τ	1
α	0.5
Hidden layers	1
Hidden units	20
Total weights	441
Hidden Activation	Tanh
Output Activation	Linear
Training set size	1000
Validation set size	1000
Learning method	SGD
Epochs	200
Mini batch size	40

The 50 trials produced 50 error traces, each 200 points long. The MOHN experiments were simpler. A fixed structure of first and second order weights was chosen and single SGD learning trials were performed with the learning rate set to each of {0.1,0.4,0.8,1}. In this case, a learning rate of 1 was found to learn most quickly, so larger rates were tried but not found to improve performance. The parameters for the MOHN experiment described below were:

Hyperparameter	Value
Structure	Fully connected at orders 1,2
Number of weights	211
Learning rate	1
Learning method	SGD

4.3.4.2 *Results*

Figure 4.2 shows the trace of the validation error during 50 attempts at learning the concatenated XOR function with an MLP and a MOHN. There is a small variation in the error trace for the MOHN, which is not due to random starting points—all networks start with weights at zero—but is due to the fact that each training set is generated at random. Note that there are no traces that indicate a local minimum for the MOHN; all go to zero error. In contrast, the MLP

Measure	MLP	MOHN
Mean final validation error	0.05	0
Final validation error s.d.	0.002	0
Average epochs	200	6

Table 4.3: Mean and standard deviation of error and average number of epochs to completion of 50 MLPs and 50 MOHNs trained on a 20 input version of the concatenated XOR function.

trace shows longer learning times and a number of attempts that have not reached a zero error after 200 epochs.

Table 4.3 show a summary of the results across the 50 trials of each method. The MOHN stopped training when the error was less than 0.000001. The same target was used for the MLP, but was never reached, so training stopped at 200 epochs.

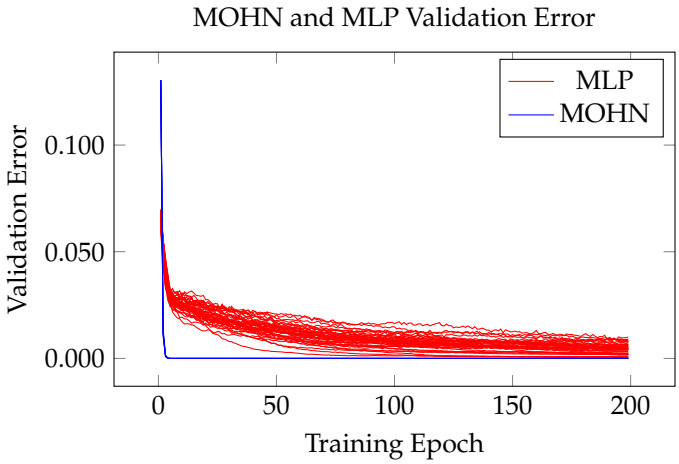

Figure 4.2: Validation error trace of 50 attempts at learning a concatenated XOR function with a MOHN (blue) and an MLP (red). The MLP learns more slowly, with more variance and with fewer runs reaching the minimum error.

One aspect of the experiments described above that was not addressed was the question of required data set size. Data was generated from the training function in a constant stream. With 1000 examples per training epoch and 200 epochs, the MLPs were presented with 200,000 different training examples. We know that a MOHN that contains the necessary weights can learn a function correctly with as many noise free unique training examples as there are weights in the MOHN (see section 4.12.2.2 for experimental evidence of this). A final set of experiments using concatenated XOR were run to establish the smallest data set that would reliably allow an MLP to reach the error levels described above.

Possible training set sizes from 500 (just over the number of weights in the 20 hidden unit network) to 2000 in steps of 500 were tested. A limited grid search was carried out, informed by the previous experiments. Tests were performed with hidden activation functions chosen

Training Points	Average Error
500	0.014
1000	0.0035
1500	0.0027
2000	0.0025

Table 4.4: Average test error over 50 trials of an MLP learning a 20 input concatenated XOR function from data sets of sizes from 500 to 2000.

from {Logistic,Tanh} and learning rate from {0.1,0.2,0.3}. The logistic activation function with a learning rate of 0.3 was found to be most effective. The only difference between the networks in this experiment and the one described above, then, is the choice of activation function on the hidden units (logistic in this case).

Fifty trials were repeated, training an MLP on a new randomly generated training set of sizes in {500,1000,1500,2000} with the number of epochs chosen to ensure that each network had seen the same number of examples (and so, had the same number of weight updates). The total number of weight updates allowed to the networks was increased to one million to give them a chance to reach a minimum.

Table 4.4 shows the the average error over 50 trials after 1 million epochs for data sets of each of the the four chosen sizes. The largest jump is from datasets of size 500 to 1000, after which the additional gains are much smaller. The MLP contained 441 weights, so required around twice as many data points as there were model parameters. The MOHN compares favourably, only needing 1 noise free training example per model parameter.

4.3.4.3 *Learning Randomly Structured Functions*

The concatenated XOR function was chosen to illustrate the difference between the convex cost function minimisation carried out by a MOHN and the non-convex function that an MLP attempts to minimise using gradient descent. With only second order connections, it is clearly an easy function for the MOHN to learn. This experiment compares MLPs and MOHNs when attempting to learn a function that is designed to be very difficult for a MOHN to learn, the randomly structured functions described in section 4.1.1.7.

Randomly structured functions are very difficult for the MSDA to learn as there are no patterns in the connectivity for it to exploit. MSDA is designed on the premise that the connectivity pattern in the function being learned is not uniformly random. These experiments address the question of how well a MOHN with the wrong weights performs and how much better (if at all) an MLP can do. Two approaches to building a fixed structure MOHN to learn random structure functions were investigated. MOHNs with only low order weights were compared with MOHNs with random weight structure.

In the first set of experiments on random structured functions, a single function over 8 inputs and containing 20 weights was generated and learned using an MLP as described below. Training data of 100 unique examples were generated by picking uniformly random input patterns and evaluating them with the random structure function. A validation set of 100 examples was also generated in the same way, ensuring that no examples in the validation set also appeared in the training data. The MLP hyperparameters were searched using grid search but the initial results suggested that the variation in error due to random weight starting points was greater than the variation due to hyperparameter choice. A second grid search was performed but this time each configuration of hyper-parameters was used to train ten different models and the average performance across each set of 10 was recorded. The hyperparameter sets used in the grid search were as follows:

Hyperparameter	Grid Set
η	{0.1,0.2,0.4}
α	{0.8,0.5}
Hidden activation	{Tanh, Logistic}
Number of hidden units	{6,12,24}
Number of hidden layers	{1,2,3}
Random weight range	{0.01,0.1,1}

All networks had linear output units and were trained using SGD in which weights were updated once for each training pattern in turn (i.e. batch size was 1) over 20,000 epochs. From the results of the grid search, a network with 3 layers, each of 4 units was chosen, making a network with 68 weights for 100 training examples. The full set of parameters was:

Hyperparameter	Value
η	0.1
Learning rate decay	None
α	0.5
Hidden activation	Logistic
Hidden Layers	3: 6,6,6
Random weight range	0.26
Training epochs	20,000

Table 4.5 shows the average correlation between the correct function output and the model output over 200 random structured functions. The difference between the MLP validation correlation and that of the 1st and 2nd order MOHN is not statistically significant (p=0.56 on a

Method	Average Test Correlation
MLP	0.203
1st and 2nd Order MOHN	0.211
Random weight MOHN	0.049

Table 4.5: Average correlation between the correct function output and the model output over 200 random structured functions for an MLP and two differently structured MOHNs.

paired, two tailed t-test). MOHNs with first and second order connections significantly out performed MOHNs with the same number of randomly assigned weights. Of course all of the models performed very poorly. It was expected that the MOHNs would perform poorly as they all had the wrong weights to learn the functions. These experiments have shown that MLPs find such functions equally difficult to learn.

4.3.5 *Learning Random Pyramid Functions*

In this section, the MOHN regression learning rules are compared with each other and with a standard multi layer perceptron (MLP). To compare the learning rules separately from the structure discovery algorithm, these experiments are on MOHNs of fixed structure, and no attempt is made to optimise the structure. For the MLPs, however, the number of hidden units is one of the hyperparameters that are searched prior to generating the training results.

The multiple pyramid function of equation 4.4 was used for these tests as it is possible to generate arbitrary functions containing a chosen number of turning points at random locations. This allows the different methods to be tested across many different functions of varying degrees of complexity.

4.3.6 *Varying the Number of Inputs*

The multiple pyramid function was used to test the training speed of each of the different MOHN learning algorithms on noisy data, compared to a standard MLP with SGD learning. The SGD method given in algorithm 2 initialises the network weights using the parity rule of equation 3.12. In these experiments, that approach is compared to the same algorithm without the initial parity setting step. For the purposes of comparison here, SGD with the parity step is referred to as SGDp.

4.3.6.1 *Experimental Setup*

THE DATA A set of experiments were conducted to compare the speed of each of the MOHN learning algorithms with an MLP. In each experiment, a function with between 15 and 60 inputs was built by randomly placing four targets using the Hamming similarity function of equation 4.4. The output from the function had normally distributed noise added to it with a mean of zero and a variance of 0.01. A data set was then generated containing n^2 samples (n is the number of inputs) from the function, representing approximately twice the number of weights in a MOHN with first and second order connections. A validation set was used to control early stopping in the SGD algorithms and an independent test set was generated for each model.

THE MODELS Each data set was stored and used to train five different models. Four were MOHNs, each trained with a different algorithm and the fifth was an MLP. The MOHN training algorithms were SGD, SGDp, OLS, and the lasso. Each MOHN was fully connected at orders 1 and 2 and no structure discovery took place. No regularisation was applied to the OLS model.

The stopping criterion for the SGD trained models was established by training a MOHN using OLS and measuring the validation error. This was then used as the target validation error for the other methods. One would expect some of them to perform better than OLS on validation, but for measuring learning speed, it is a convenient target. The model hyperparameters were optimised once, as described below, and the optimised parameters were then used to train 45 models (varying in size from 15 to 60 inputs) 50 times to gather data on both the mean and the variation in training time for each method.

The MLPs used in this experiment had a single hidden layer and were trained using SGD and regularised using dropout. As the functions varied in size from 15 to 60 inputs and no two functions were the same, optimisation of the hyperparameters was done across a sample of functions of each size. Some hyperparameters were set using a function of the number of inputs in an attempt to optimise over all the possible functions to be learned. This can become a time consuming process so only the following hyperparameters were searched:

Parameter	Description	Range
h	Ratio of hidden units to inputs	$\{0.25, 0.5, 1, 2, 3\}$
lr	Product of η and network size ($\eta = lr/n$)	$\{1, 2, 3, 4\}$
sw	Starting weight range	{0.25,0.5,1,2,4}
d	Dropout rate	{0,0.2,0.4,0.5}

Each combination of hyperparameters was applied to 10 different randomly generated pyramid functions and the model with the lowest average test score was selected. The hyperparameters from that model were used in the next step of the experiment. As the MOHN

structures were fixed, there were few hyperparameters to explore. For lasso, the λ value was the only hyperparameter and for SGD, only learning rate was explored.

4.3.6.2 *Results*

Figure 4.3 shows the mean training time with error bars at one standard deviation over 50 trials. For these data, the parity initialisation step moved the weights of a MOHN very close to the point where the validation error stopping criteria was met and required a very small number of further SGD steps to reach it. Consequently, the time taken to reach the stopping criterion was lowest across the trials. It also grew at the lowest rate of all the methods compared. SGD with weight values initialised to zero was the worst performer, with the longest training times and the most variance. The MLP took longer to reach the validation error set by OLS than all of the MOHN methods except SGD.

Once each model had been trained and validated, the test data was evaluated to provide a final measure of model quality. Figure 4.4 illustrates how test error varied with number of inputs for the different learning methods. OLS and SGD both follow a similar pattern, due to the fact that they are minimising the same unregularised cost function. SGD has a slightly better test error than OLS due to early stopping. The MLP performs better than the unregularised OLS and SGD trained MOHN, but the MOHNs trained with parity initialised SGD or lasso reach a lower test error.

4.3.7 *Error Descent Rate*

The behaviour of the training and the validation error during training is of interest as it gives a useful insight into the generalisation and over fitting behaviour of the learning algorithm as it progresses. These were investigated experimentally by training a MOHN and an MLP on a function with 30 inputs and 4 randomly placed local maxima. The hyperparameters discovered for the previous set of experiments were used for the MLP training, that meant an MLP with 15 hidden units in a single hidden layer trained with a learning rate of 0.3, a momentum rate of 0.8, a drop out rate of 0.2, trained using SGD (i.e. batch size of 1). Weights were randomised within r and $-r$ where $r = 4\sqrt{6/(fanin + fanout)} = 1.75$, as suggested in [13] and found to be optimal during the grid search of the previous experiments.

MOHNs were built with a fixed structure of first and second order weights only (a somewhat arbitrary design, aimed at keeping the number of weights similar to those in the MLP and motivated by the fact that a 30 node Hopfield network should be expected to be able to store 3 to 4 local optima [95]). The purpose of this experiment was not to find the optimal MOHN, but to study the behaviour of a MOHN of reasonable structure as it learned.

Fifty different functions were generated, each with different randomly placed local maxima and each was learned as described above based on a training sample of 600 randomly generated

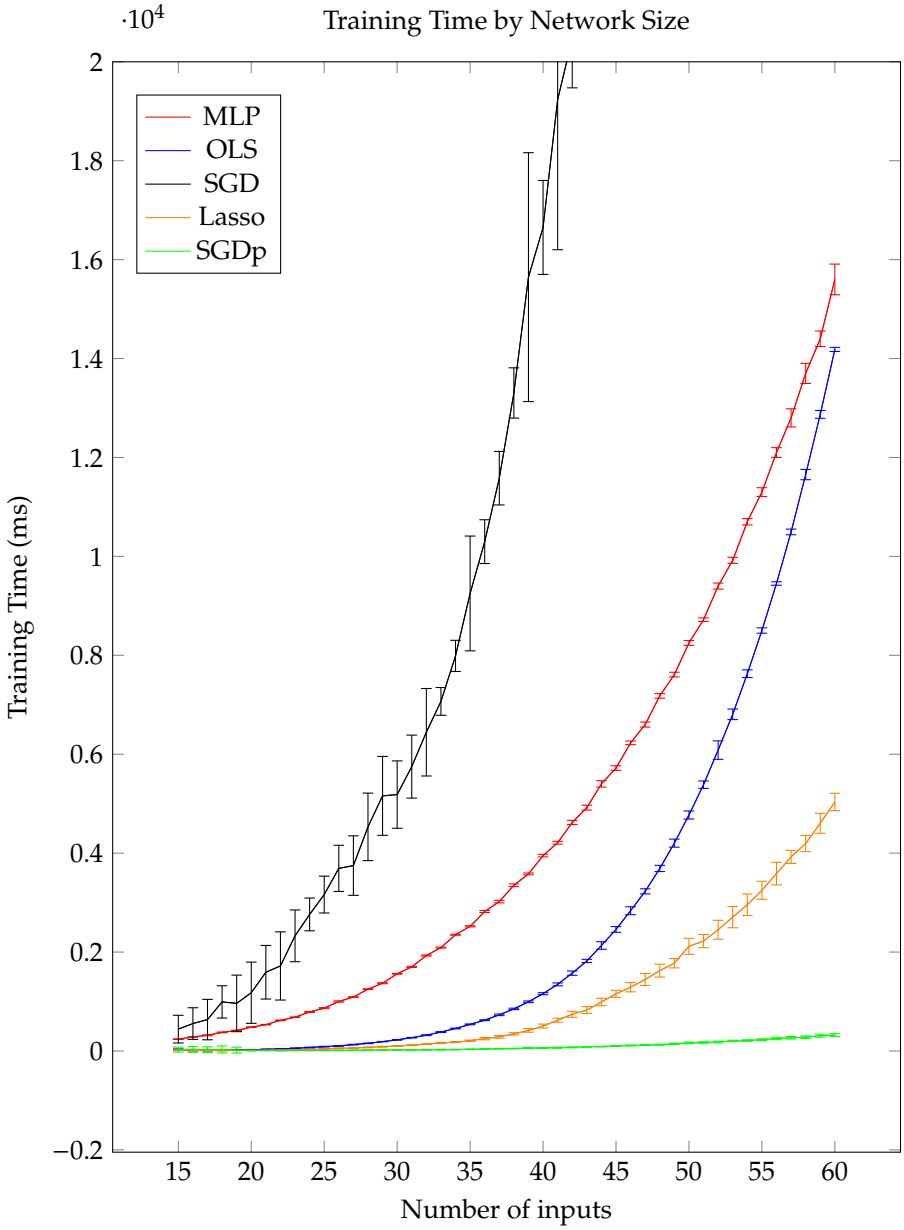

Figure 4.3: The mean and one standard deviation of training time for an MLP and four different MOHN learning rules as network size varies. All models were trained on noisy data from functions with four randomly placed local maximum. Each data point is calculated from 50 trials.

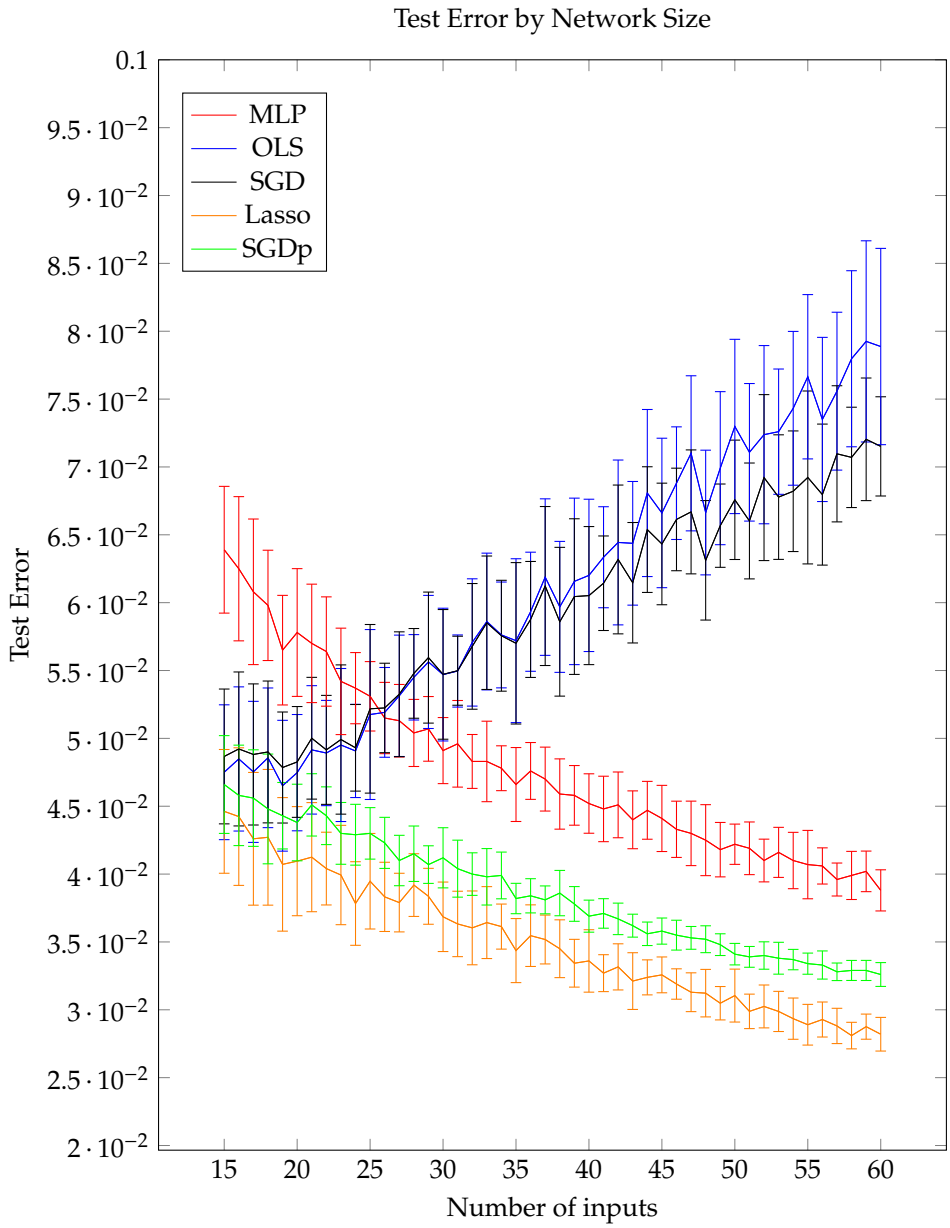

Figure 4.4: The mean and one standard deviation of test error for an MLP and four different MOHN learning rules as network size varies. All models were trained on noisy data from functions with four randomly placed local maximum. Each data point is calculated from 50 trials.

input patterns and their associated outputs. A further 600 data points were used to track validation error (which, due to the fact that this data was not used to influence any training decisions, was equivelently the test data).

Figure 4.5 shows the average training and validation error on each pass of the training data from 50 repeated trials. Several points may be noted. The MLP's train and validation error values are always very close together, whereas the MOHNs fit the training data very well but show a similar validation error to that of the MLP. The MOHN trained on SGD alone trains slowly and fails to reach the validation error of the MLP, as expected from the results of the previous experiment.

By initialising the MOHN weights with the parity based values and then learning with SGD, however, we see that the validation error starts lower than the validation error that either the MLP or the SGD MOHN ever reach, drops for a small number of steps, and then starts to climb. The SGD MOHN and the Parity-SGD MOHN both converge in terms of both training and validation error, as one would expect as they are optimising the same convex function. The parity based weight initialisation calculates the weight values independently, and as a result introduces bias. In the case of this experiment, that bias has put the weight vector in an almost optimal point in terms of generalisation.

4.3.8 *Conclusion*

The different MOHN learning rules were compared with each other for fixed structure MOHNs and with MLPs. Using a fixed structure fixes model bias so that the learning rules can only differ in terms of estimation bias. As the squared error cost function of the weights of a MOHN is convex, we would expect any unbiased gradient descent method to approach the OLS solution, and this was shown to be the case. Regularisation (estimation bias) was introduced using lasso learning and early stopping of the SGD algorithm. Additionally, the parity weight initialisation was shown to speed SGD learning and, when coupled with early stopping, introduce model bias.

For most problems, however, using a fixed structure is not a feasible approach as the important weights are not known and a fully connected structure, even at the lower orders leads to a very large model. Structure discovery is designed to overcome this problem and is discussed next.

4.4 Structure Discovery Experiments

This section presents the results of some experiments using the MOHN structure discovery algorithm (MSDA).

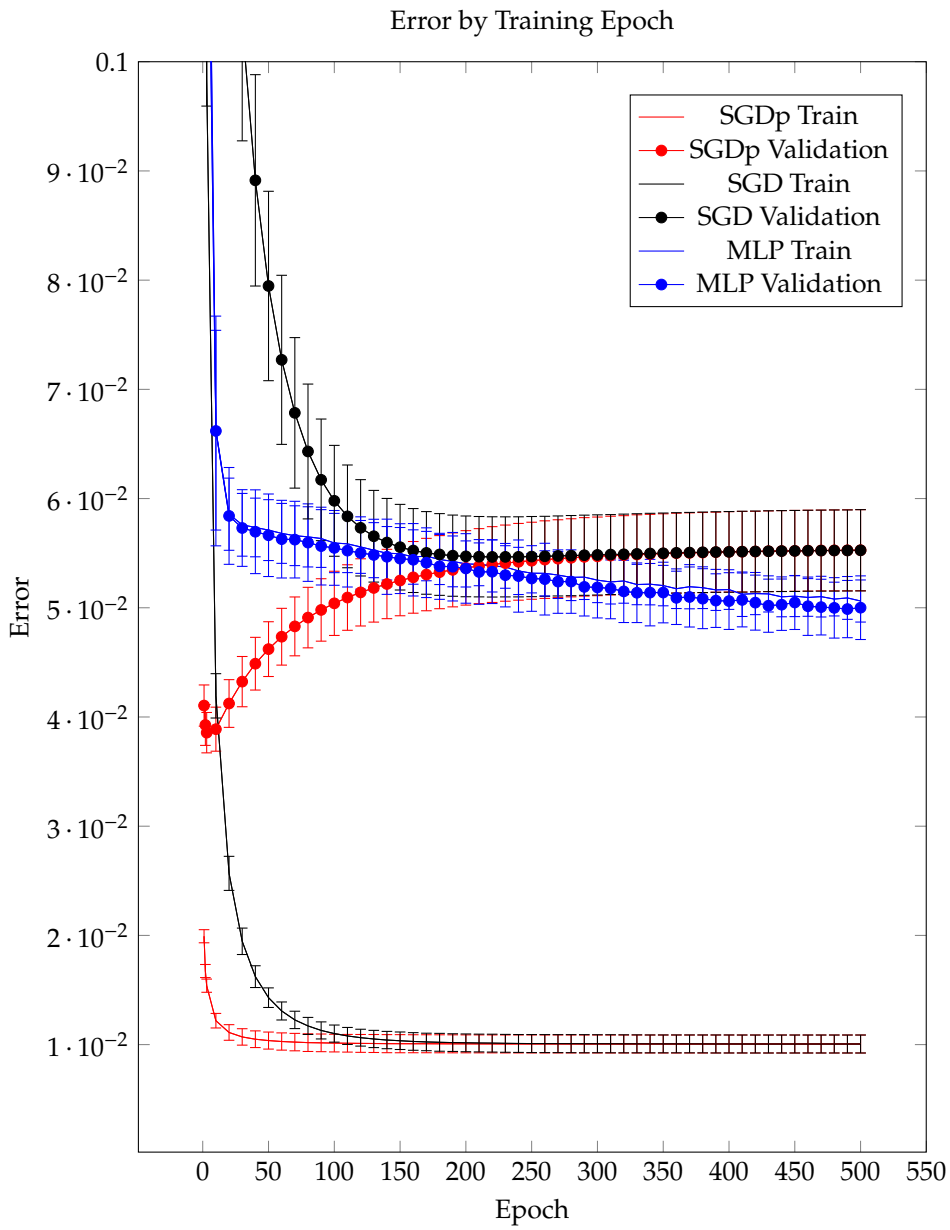

Figure 4.5: Training and validation error during training of an MLP and a MOHN, the latter using SGD with and without a parity weight initialisation.

4.4.1 *Graph Colouring Function*

The first experiment uses the graph colouring problem, which involves searching for a way to colour the nodes of a graph so that no two connected nodes share a colour, using a limited palette of colours. The encoding and cost function of the graph colouring problem is described in section 4.1.1.6. Although this is a problem requiring a search heuristic, the main purpose of using it in this section is to test the MSDA's ability to find the correct structure and parameters to represent the cost function of a graph colouring problem based on samples from that function.

4.4.1.1 *Experimental Setup*

This set of experiments compares the MSDA with an MLP on the task of learning a randomly generated graph colouring function. Each trial in this experiment involves a graph of ten nodes with twelve edges added at random, but in such a way that would permit a four colouring (no node has more than two neighbours).

The MLP input consisted of 40 neurons (ten groups of four) and the target output was the fitness value of the given input pattern when evaluated using equation 4.9. Each resulting function was sampled repeatedly to produce on-line training data and tested periodically on new randomly generated validation data. A little trial and error revealed that testing every 100,000 epochs gave a useful idea of the progress being made, and this is the interval used to generate the validation error results in figure 4.7. The MLP hyperparameters were chosen using a grid search over the following values:

Hyperparameter	Grid Set
η	{0.05,0.1,0.2,0.3}
α	{0.8,0.5, 0.1}
Learning rate decay	{0,0.5,1}
Hidden activation	{Tanh, Logistic, ReLU}
Number of hidden units	{20,40,80}
Number of hidden layers	{1,2}
Random weight range factor	{0.01,0.1,1}

The MLPs were trained using SGD with no batch learning. Training during the grid search was limited to 500 epochs (an epoch being the presentation of 100,000 randomly generated training points) and the validation error at the end of that period was used as the measure of success for the hyperparameter set. As a result of the preliminary search, the hyperparameter search space was narrowed to 40 or 80 hidden units, in 1 or 2 layers, a learning rate of 0.05,0.06,0.07 or 0.1, a momentum value of 0.8, no learning rate decay, and a random weight range of 0.1. A second

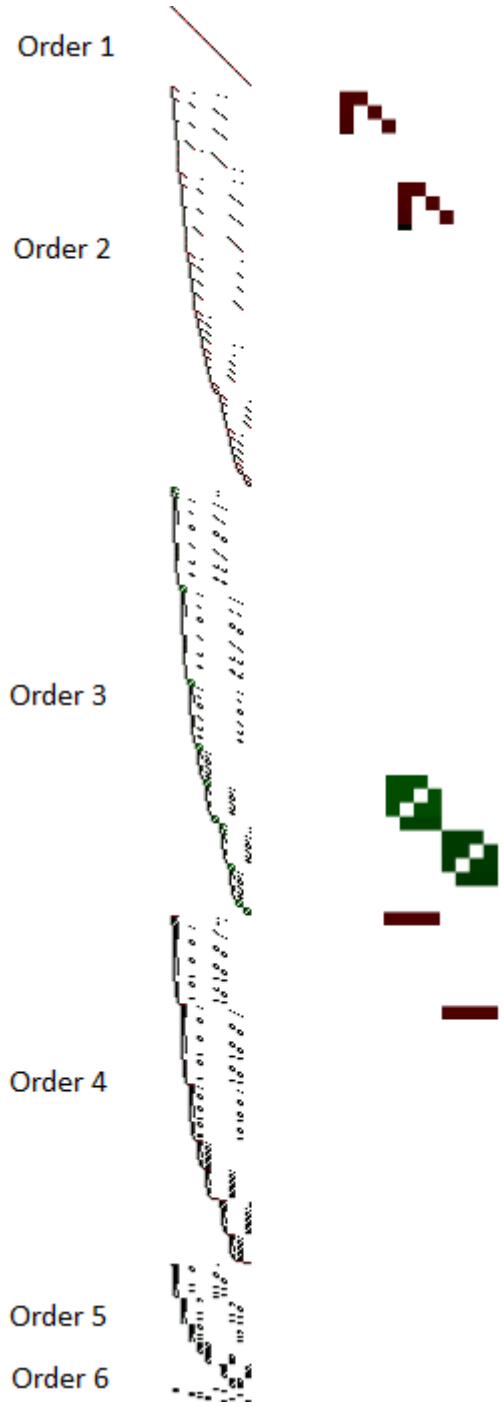

Figure 4.6: The weights from a MOHN trained on samples from a graph colouring problem fitness function. The enlarged examples show parts of the learned implementation of the 1-of-4 encoding used to represent the colour of a node.

grid search across these values led to a final choice of:

Hyperparameter	Value
η	0.06
α	0.8
Learning rate decay	0
Hidden activation	Logistic
Number of hidden units	80
Number of hidden layers	1
Random weight range factor	0.1

100 MLPs were then built, trained and tested as described above, with test error being recorded at the end of each epoch. The mean and standard deviation of the test error at each epoch across all 100 trials was then calculated.

When training the MOHN, the hyperparameters that control some aspects of the MSDA were searched. As the error profile for the MLP had already been established, the MOHN search was limited to attempting to find a model that performed no worse on validation data than the MLPs. The hyperparameters that were considered were the number of training epochs made through the data between weight removal and addition (i.e. per iteration), and the critical p-value that determined which weights were removed. A grid search was not considered necessary. The number of training epochs was chosen by looking at the rate of decrease in the test error from one epoch to the next and choosing the point where it became flat (defined as an average change in absolute validation error over five epochs of less than 0.001). This led to a choice of ten epochs per iteration of the algorithm. Critical p-values of $\{0.05, 0.1, 0.2, 0.5, 0.7\}$ were explored by training one MOHN per candidate value for 500 iterations and comparing the final validation error. A critical value of 0.5 was chosen from these results.

Other hyperparameter decisions for the MOHN were made as follows. Assuming nothing is known about the weight orders required, the default assumption that lower orders should be tried first was made so the weight order picking distribution was initialised with a mode of 1 and $\lambda = 1$, which causes the probability of weights with orders over 4 to be very near zero.

As with the MLP, 100 graphs were generated and used to produce training data for 100 MOHNS, each trained for 500 iterations with 10 training epochs per iteration. The validation error at each epoch was recorded for each network and then averaged over the 100 trials. For the MOHN, this involved recording the validation error just before weights were removed (at the point where the new weights had been added and a set of SGD iterations had been performed). This smooths the error profile as the jumps caused by adding and removing weights are not recorded. As the MOHN made ten passes through the training data per iteration, the error from the MLP was extracted at intervals of ten so that the errors were directly comparable.

4.4.1.2 *Results*

Figure 4.7 shows the mean and two standard deviation range of the validation error for the MLP and the MOHN as training progressed. The MOHN is consistently faster to learn and more accurate than the MLP.

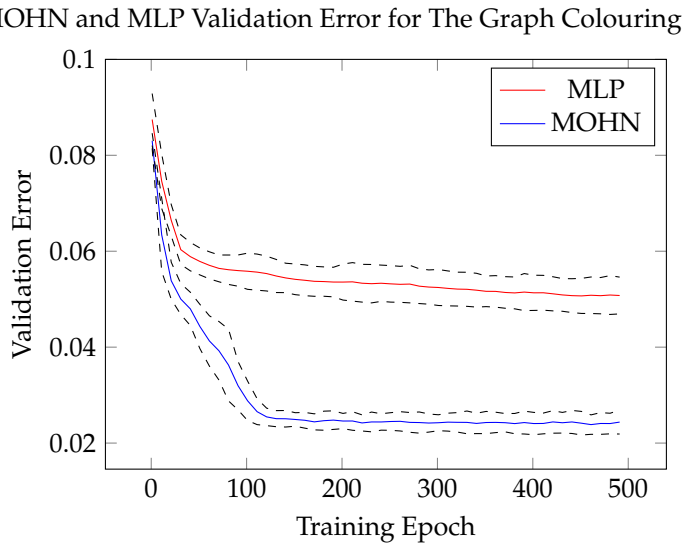

Figure 4.7: Mean and two standard deviation range of the validation error over 100 trials learning the graph colouring problem fitness function with an MLP (top, red line) and a MOHN using MDSA and SGD (lower, blue line).

Having learned the graph colouring function, acceptable colourings were found using random restart hill climb on the resulting MOHN. Figure 4.8 shows an example solution generated by learning a graph function and then settling the resulting network. Figure 4.6 shows the structure of the network with some of the detail that represents the 1-of-4 coding imposed by the function in equation 4.9 enlarged. The groups of four show negative second order connections, positive order three connections, and a negative connection at order 4.

Of course, learning a model and then using that model to perform the optimisation is not an efficient way of solving the graph colouring problem, but the example is used because the MOHN solves the problem blind. If the structure of the problem were not known, and samples from the fitness function were the only guide, then the MOHN would be appropriate as it reveals the structure to a human observer. As graph colouring problems also have many equal optima, they also serve as an example of a situation where a fitness function model may be preferable to repeated heuristic search as the model is capable of producing multiple solutions based on random start points.

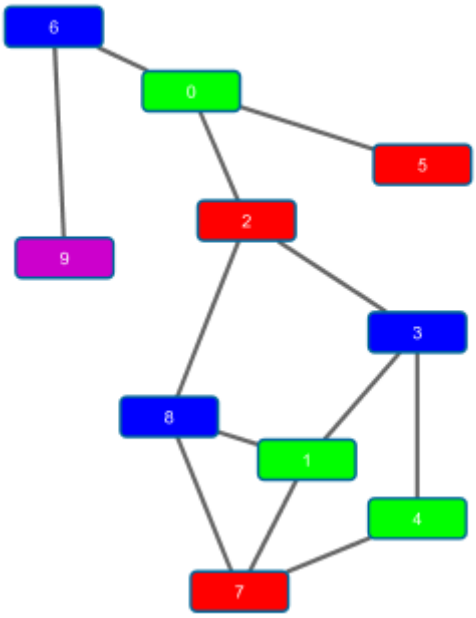

Figure 4.8: An example solution of a small graph colouring problem created by learning the function with a MOHN and then settling the MOHN to an attractor.

4.4.2 *Comparing The Lasso and SGD Learning During Structure Discovery*

This section compares SGD and the lasso learning rules used in the MSDA. The design justification for using SGD after the addition of new weights is that the existing weights should already be close to their desired values so intuition suggests that this will be faster than using the lasso across the whole network. This section presents some experimental results comparing the two using the k-bit trap function. Speed and accuracy (in terms of root mean squared error) were compared over 100 runs of the structure discovery algorithm as it attempted to learn the structure and weights of a 5-bit trap repeated 6 times over 30 input variables.

The MSDA was run repeatedly with the following hyperparameter settings:

Hyperparameter	Value
Training examples	10,000
Initial number of weights	2,500
Weights added per iteration	1,200
SGD epochs	20
SGD critical p	0.5 dropping by 0.1 every 10 iterations
SGD learning rate	0.3
Weight addition iterations	50
Used weight list emptied every	15 iterations
Weight order distribution	Mode=1, $\lambda = 1$

Algorithm 6 was used when the learning method was lasso and algorithm 5 was used when the learning method was SGD.

Figure 4.9 shows validation error by iteration of the structure discovery algorithm, averaged over the 100 trials. The lasso consistently achieved a lower error, but took on average over ten times as long to compute as SGD learning. Figure 4.10 shows the median, inter quartile range and full range of the time taken by SGD and the lasso to find the correct structure for the same problem.

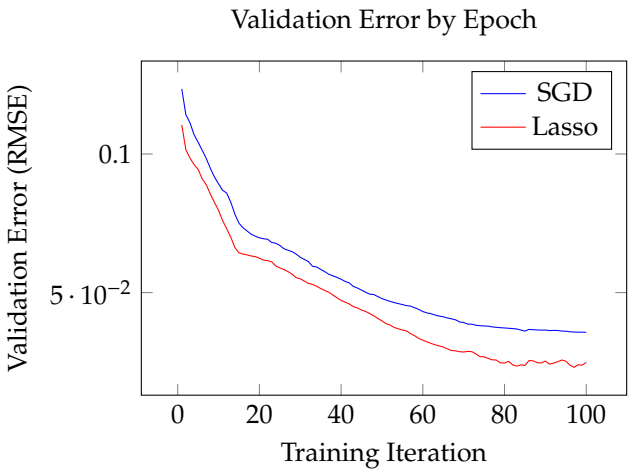

Figure 4.9: Validation error during structure discovery using SGD and a t-test to remove weights (top blue line) and the lasso to learn and remove weights (lower red line). Both lines represent an average over 100 trials.

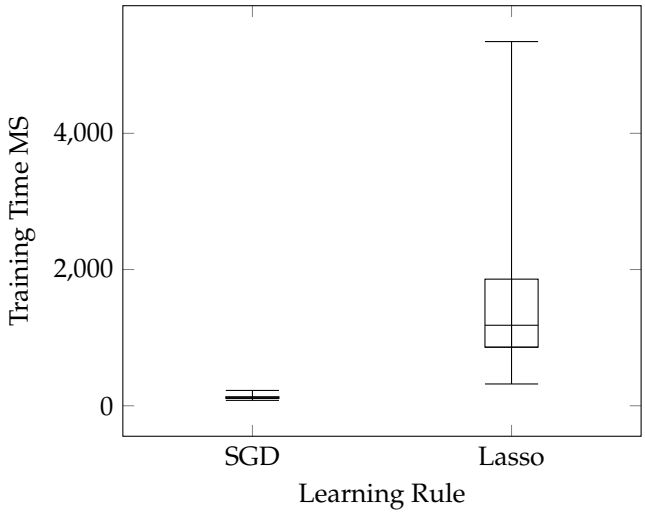

Figure 4.10: Median, inter quartile range and full range of the time in milliseconds taken by SGD and the lasso to find the correct structure for the 5-bit trap over 30 inputs.

4.4.3 *Learning Under Noisy Conditions*

The examples given so far have involved noise free samples from known functions. This is motivated by the use of such models as fitness function surrogates, in which cases noise free samples are often available. The MOHN learning rules (SGD, OLS and the lasso) are all known to perform under noisy conditions, but the required quantity of data increases with the level of noise. The presence of noise also affects the performance of the structure discovery algorithm, as larger samples are required to allow the correct relationships to be found.

A set of experiments was performed to test the efficacy of the MOHN structure learning algorithm under noisy conditions. The experiments also sought to discover the ability of the MOHN to scale to larger problems.

4.4.3.1 *Experimental Setup*

In the first set of experiments in this section, a 4-bit trap function was used to score randomly generated data points. Data was generated continuously, rather than taken from a sample of fixed size. Functions with 4-bit traps ranging in size from 16 to 76 inputs were generated (i.e. numbers of traps from 4 to 19) with normally distributed noise with a mean of zero added to the outputs from the function. In different trials, noise was set to have a variance of 0 (no noise), 0.01 and 0.05 in turn. Figure 4.11 shows a small sample of the function output plotted against the noisy equivalent with the variance of the noise at 0.05 to give the reader an indication of the size of the noise.

The MOHN structure discovery algorithm was trained on data sampled from a function of each size with its parameters set as follows. The weights added to the MOHN were limited to

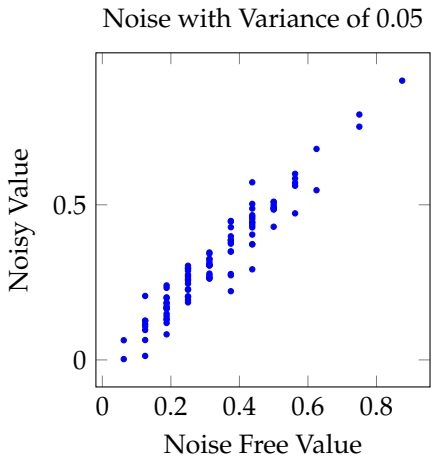

Figure 4.11: A sample of outputs from the 4-bit trap function, plotted against the noisy values used to test the MSDA. Noise is normally distributed with a mean of zero and a standard deviation of 0.05.

order 5 (one above the known weight order for this task). Each iteration of the MSDA used a sample of data equal in size to the number of inputs to the model times 200, selected at random and scored with equation 4.7 and with noise added to the output according to the current experimental settings. A single training iteration involved 20 passes through the training data. At each iteration of the MSDA, the number of weights added to the model was equal to the number of training samples divided by eight. Experimentation with the p-value used to discard weights found that very high values were needed as noise was added, so the critical p-value started at 0.9 and descended to 0.3 as the learning progressed. The SGD learning rate was fixed at 0.3.

4.4.3.2 *Results*

The number of possible weights in a MOHN grows exponentially with the number of inputs but the number of possible weights up to a given order, o grows more slowly, at a rate given by $\sum_{k=1}^{o} \binom{n}{k}$. The time taken for each MOHN to learn its target function is plotted in figure 4.12 along with the size of the search space of weights up to order 4. The time taken by MSDA to find the correct structure grows in line with the search space size. The MSDA is searching the larger space of weights up to order 5, but the probability distribution over weight orders prevents the order 5 weight space being searched very far.

At the end of each trial, the error made by the final model was assessed. Possible sources of error are model bias caused by the MSDA finding an insufficient structure, estimation bias caused by the SGD algorithm stopping short of the error minimum and the residual error due to the noise added to the training data. Estimation bias was removed by learning the training data with the structure generated by the MSDA using OLS, which is an unbiased estimator. The remaining error was compared to the level of noise that was added to the data (average

error of 0.01 or 0.05). Across all the trials, the average validation error from the final structured MOHN, trained with OLS was within 0.0001 of the level of noise added to the training data. This suggested that the only source of error was the noise. A final test was carried out in which a new training set with as many examples as there were weights in the model to be tested was generated without added noise and used to estimate a new set of weights for each structure discovered by the MSDA. Another noise free validation set was then generated and evaluated by that final MOHN. In each case, the validation error was zero, showing that no model bias was present in the structure discovered by the MSDA.

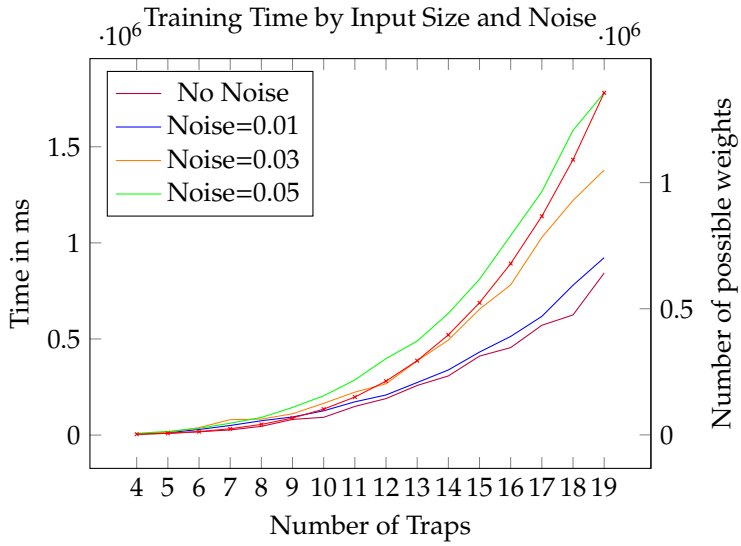

Figure 4.12: Mean training time in milliseconds over 25 runs learning the 4-bit trap function for network sizes from 4 to 19 traps and at four different levels of noise. The red line with x markers shows the size of the search space up to order 4 connections.

4.5 Content Addressable Memories

MOHNs act as content addressable memories by moving to an attractor state from any starting state, as described in section 3.2.1. The attractor states represent local minima in the energy function, which is limited in the number of minima it can represent by restrictions on the model.

Attractor states can be fixed by either training the network on the patterns they represent or by learning the energy function in which they are minima. In the latter case, the identity of the patterns may not be known, but attractors can be inferred by learning the function from samples of data. This learning process can be geared towards only learning the attractors, or to minimising the error between the model and the real function behind the data. These three modes of learning each have an associated learning rule and are addressed in the following sections.

4.5.1 *Hebbian Learning*

The Hebbian learning rule of equation 3.3 allows a certain number of memories to be loaded into a MOHN. The result of loading patterns into a MOHN is that the energy function has local minima at states represented by each of the target patterns. Any set of patterns can be loaded one at a time up to the point where the most recent pattern causes one or more of the existing patterns to no longer occupy a local minimum. For a given set of patterns, the number stored before a memory is lost defines the network's capacity.

This section investigates the capacity of MOHNs of various sizes for storing sets of random patterns. The traditional method for experimentally testing the capacity of a Hopfield network ([122] for example) is to load random patterns one at a time and then test whether the network still maintains all of the patterns learned so far as attractors. The process is given in algorithm 16.

Algorithm 16 Testing the capacity of a MOHN

for a MOHN, $M = (X, \mathbf{W})$
$\mathbf{P} = \emptyset$ ▷ Start with an empty pattern set
$\omega_j = 0 \, \forall j$ ▷ Set all the weights in the MOHN to zero
repeat
  Generate $x \notin \mathbf{P}$ ▷ Generate a random pattern that is not in the pattern set
  $\omega_j = \omega_j + \frac{1}{m} \prod_{i \in \mathbf{I}_j} x_i \, \forall j$ ▷ Learn x
  $stop = false$
  for all $x \in \mathbf{P}$ **do**
    $X = x$ ▷ Set the inputs to each pattern in the list in turn
    $X = \text{HC(M)}$ ▷ Update the neurons in fixed order
    if $X \neq x$ **then** ▷ If any neuron value changes
      $stop = true$ ▷ Attractor is destroyed and capacity exceeded
    end if
  end for
  if $stop = false$ **then** ▷ No pattern was lost, so add new one to list
    $\mathbf{P} = \mathbf{P} \cup x$ ▷ Add the pattern to the set
  end if
until stop ▷ End when a pattern is lost

To conclude that a pattern is no longer an attractor state in a network, it is only necessary to find a single neuron that would change its value when the network is in that state. Rather than the usual method of running a network by updating the neurons in random order, the attractor test simply updates each neuron in turn in a fixed order (avoiding the overhead of randomisation) and stops as soon as a neuron changes value, indicating that the pattern is not

an attractor. When all neurons have been tested and none have changed, the pattern is proved to be an attractor.

A series of experiments to ascertain the capacity of various MOHNs were performed. Each involved repeating the process in algorithm 16 for networks of fixed structure but of varied size. For example, the experiments into the capacity of second order networks (i.e. Hopfield networks) kept the structure of full connections at order 2 fixed, but varied the number of neurons connected from 10 to 100. Each network was tested 100 times so that a range of capacity values could be found. The figures that follow show the results for networks of order 2,3,4,5 and 6 The circles show the average capacity across 100 independent trials and the error bars show the minimum and maximum capacity found. The solid red line shows the theoretical weak lower bound on capacity and the solid green line shows the theoretical lower bound, both according to Baldi et al. [141].

There are a number of things to note from the figures. Firstly, the exponential growth in the number of weights in a network as higher orders are fully connected means that the figures 4.14, 4.15, 4.16 and 4.17 have fewer points plotted. Secondly, figure 4.17 shows the results for a network connected at order 6 alone, whereas the others are connected at all orders up to the indicated maximum.

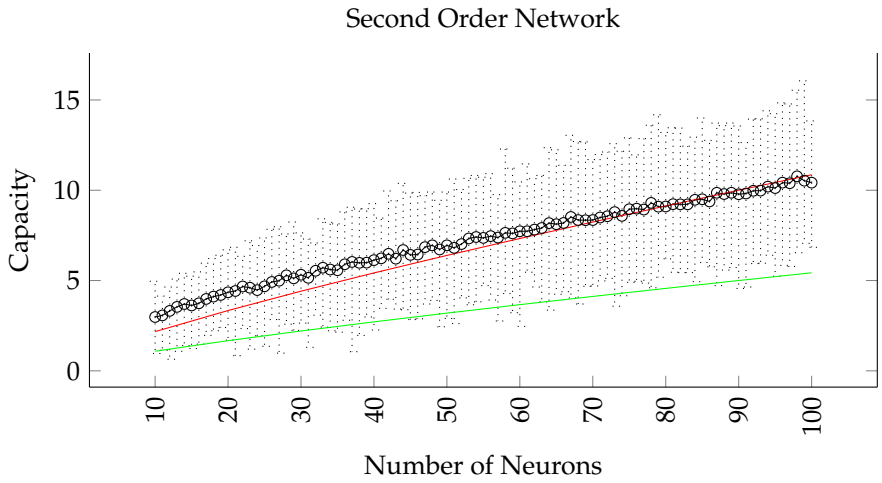

Figure 4.13: Experimental mean and range of capacity of a second order MOHN (equivalent to a Hopfield network) (circles and error bars). The minimum weak capacity, $\frac{n}{2\ln n}$ (red line) and the minimum capacity, $\frac{n}{4\ln n}$ (green line), both according to [141].

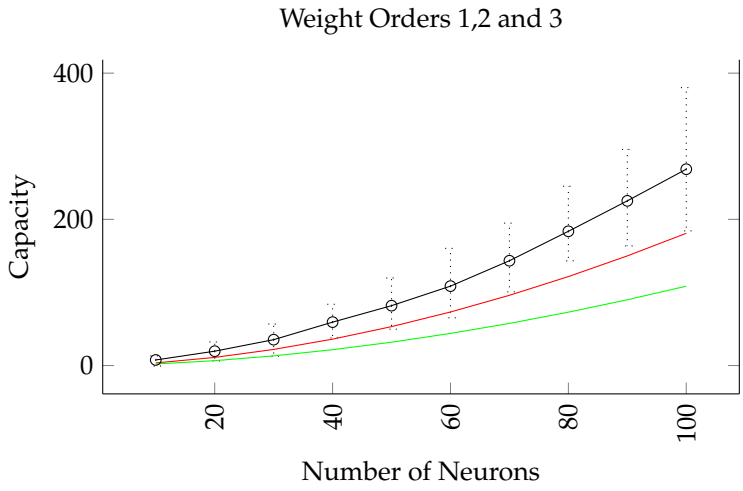

Figure 4.14: Experimental mean and range of capacity of a MOHN fully connected at orders 1,2,3 (circles and error bars). The weak lower bound on capacity, $\frac{n^2}{12 \ln n}$ (red line) and the lower bound capacity, $\frac{n^2}{20 \ln n}$ (green line), both according to [141].

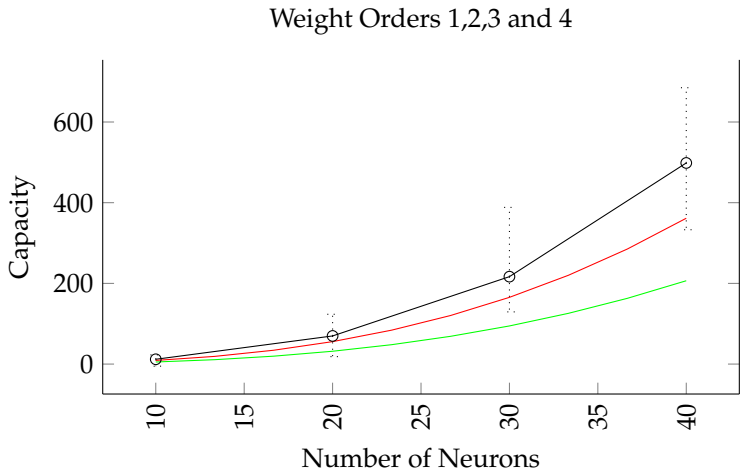

Figure 4.15: Experimental mean and range of capacity of a MOHN fully connected at orders 1,2,3,4 (circles and error bars). The weak lower bound on capacity, $\frac{n^3}{48 \ln n}$ (red line) and the lower bound capacity, $\frac{n^3}{84 \ln n}$ (green line), both according to [141].

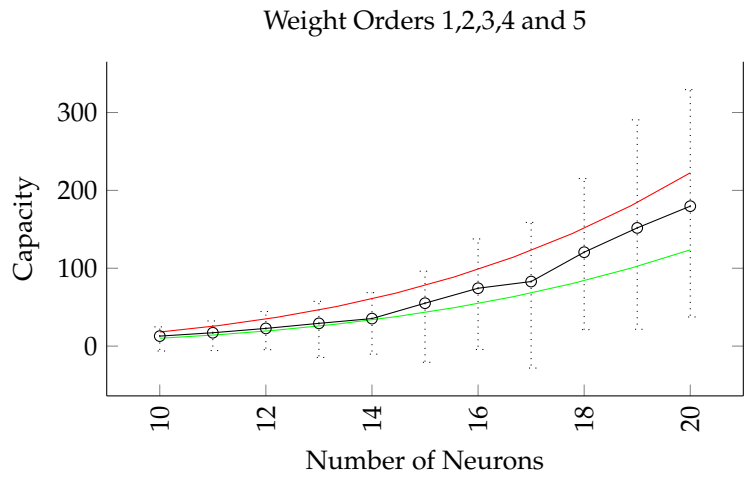

Figure 4.16: Experimental mean and range of capacity of a MOHN fully connected at orders 1,2,3,4,5 (circles and error bars). The weak lower bound on capacity, $\frac{n^4}{240 \ln n}$ (red line) and the lower bound capacity, $\frac{n^4}{432 \ln n}$ (green line), both according to [141].

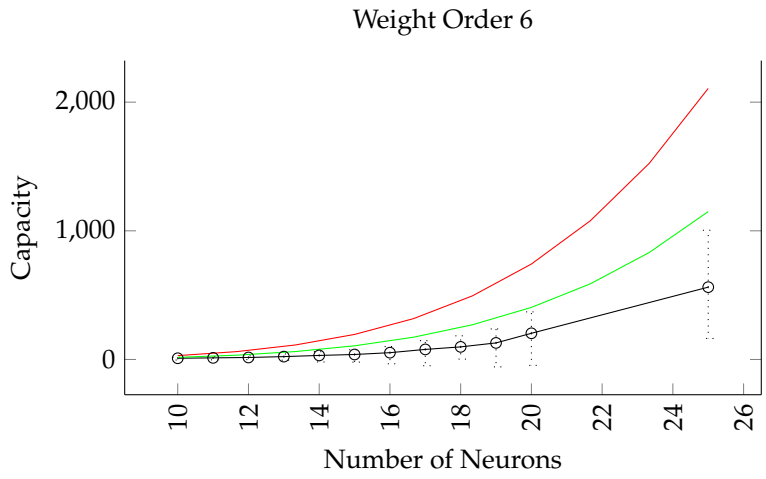

Figure 4.17: Experimental mean and range of capacity of a MOHN fully connected at order six alone (circles and error bars). The weak lower bound on capacity, $\frac{n^5}{1440 \ln n}$ (red line) and the lower bound capacity, $\frac{n^5}{2640 \ln n}$ (green line), both according to [141].

4.5.1.1 *Conclusion*

The estimates made by Baldi et. al [141] match the experimental findings from MOHNs reasonably well. Adding higher order connections increases capacity, but at a cost as the number of weights at each order grows quickly. Fully connected networks are very inefficient but sparsely connected networks need an efficient method for discovering the right weights.

4.5.2 *Improving Capacity with Structure Discovery*

To overcome the problem of exponential growth in the number of weights, a content addressable memory can be incrementally built using the structure discovery method of algorithm 8 (page 73). In this approach, weights are added to a network until it is able to store the patterns in the training set and removed if they do not contribute any improvement. The resulting network is sparsely connected, unlike those in figures 4.13 to 4.16.

4.5.2.1 *Experimental Setup*

To illustrate the storage capacity of a dynamically built MOHN, a set of patterns representing the written digits from 0 to 9 were created over 25 neurons, shown in figure 4.18. A traditional fully connected Hopfield network with 300 second order connections can only store three or four such patterns and a network with sufficient higher order weights to store them all would be very large if it was fully connected at those orders. This experiment compares the size of the smallest fully connected high order network capable of storing the target patterns with one built using the structure discovery algorithm. Patterns were learned using the Hebbian learning rule of equation 3.3 and the CAM MSDA, described in algorithm 8. 200 weights were added at each iteration of the algorithm, drawing orders initially from a Laplace distribution with a mode of 2 and $\lambda = 2$.

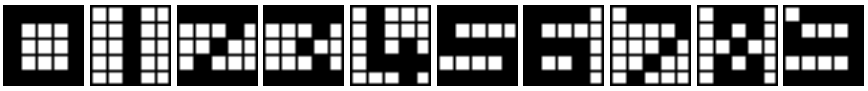

Figure 4.18: The written digits from 0 to 9 as 25 bit patterns to be used to test the dynamic structure discovery algorithm applied to a CAM.

4.5.2.2 *Results*

Firstly, static networks were tested to find the lowest order at which full connections were needed to store the patterns. Networks of 25 neurons, fully connected at all orders up to two, three and four all failed to store all 10 patterns as stable attractors. A network with all weights connected at all orders up to five was able to store the patterns. This network contained 53,131 weights. The next step is to try and discover a network that will store the same patterns in fewer weights.

The CAM MSDA, described in algorithm 8 was able to find a network capable of correctly storing all of the patterns with a total of 362 weights, which is approximately the same number as found in a standard 25 neuron Hopfield network, which would be able to store only two or three of the digits.

4.5.3 *Discussion*

In principle, this approach gives MOHNs arbitrary storage capacity, as a MOHN can represent any function in $f : \{-1, 1\}^n \rightarrow \mathbb{R}$, and for any set of non-neighbouring patterns, $\mathbf{P}$ there exists a function in which each member of $\mathbf{P}$ is a local maximum. Of course, some functions may be difficult to discover the correct structure for, and some may require so many weights that a solution is impractical, but in principle, MOHNs can store arbitrary pattern sets. If $\mathbf{P}$ contains neighbouring patterns (two patterns are neighbours if there exists an input, X_i such that flipping its value, $X_i = -X_i$, switches from one pattern to the other), then the neighbours will form a plateau where the output from the function is the same for all points. Whether these states can be considered stable attractors is a question of definition and of the implementation details of algorithm 9.

If that algorithm only moves from a state to one with higher output, it will stay stable in the first state of a plateau that it finds. This would mean that seeding it with the target states would show them all to be attractors, but that some were not accessible from nearby states. As neurons are updated in random order, the same degraded pattern might produce a different pattern on the same plateau on repeated trials. An additional step can be added to algorithm 9 in which neighbouring states of a first found attractor state are explored if they lie at the same height. This can be done by recursively making single steps from each point on the plateau until they have all been visited.

It should also be noted that building a CAM using MSDA in this way requires all of the target patterns to be available at the same time. Adding new patterns incrementally would require new weights to be added to accommodate each new pattern, which is a refinement not considered here.

4.5.4 *Weighted Hebbian Learning*

The weighted Hebbian learning rule of equation 3.4 attempts to build a network in which the local minima in the energy function correspond to turning points in the function that generated the data. The accuracy of the energy function as a regression model is of lesser importance in these cases.

This section investigates the question of whether a network with the same structure can discover the same attractors using samples from a function that maps an input pattern, X onto the Hamming similarity between X and the local maximum closest to X. These functions are constructed based on the Hamming similarity to the closest target pattern, as described in section 4.1.1.3.

By filling a MOHN to capacity using algorithm 16, a Hamming similarity based function can be built which has turning points (attractors) at each of the learned patterns. These experiments

are designed to test whether or not a MOHN can discover all of those turning points from a sample of random data points and their associated output. It is one thing to model a function in which the turning points of the sampled function are attractors, and another to learn the function correctly. A later section investigates the question of whether a MOHN needs to learn the function correctly in order to learn its turning points.

4.5.4.1 *Experimental Setup*

The procedure followed in this set of experiments is given in algorithm 17. The MOHNs in these experiments were connected at second order only.

Algorithm 17 Testing the capacity of the MOHN Learning Rule

For a MOHN, $M = (X, \mathbf{W})$ ▷ Create a network of chosen structure
$\mathbf{D} = capacityset(M)$ ▷ Generate a set of patterns that fill M to capacity
$f(X) = Hamming(\mathbf{D})$ ▷ $f(X)$ returns the Hamming similarity to the closest member of $\mathbf{D}$
repeat
  $X =$rand$(\{-1, 1\}^n)$
  $Y = f(X)$
  M learns (X, Y) ▷ Sample from $f(X)$ and allow M to learn each $(X, f(X))$ pair
  if Test$(M, \mathbf{D})$ **then** ▷ Test to see if every member of $\mathbf{D}$ is an attractor in M
    $Learned = true$
  else
    $Learned = false$
  end if
until *Learned* or Give up

4.5.4.2 *Results*

A series of experiments using algorithm 17 showed that the weighted Hebbian learning rule was able to produce a function with attractors at each of the patterns in the set originally learned by loading the patterns with the Hebbian rule.

Figure 4.19 compares the experimentally discovered capacity of second order MOHNs compared to the theoretical capacity of a Hopfield network. Networks varying in size from 10 to 100 units were tested in steps of five units. Each network size was tested with 100 repeated trials over random pattern sets using algorithm 17. The capacity of a network for storing turning points in $f(X)$ learned from samples of $(X, f(X))$ is found to be the same as the capacity for a CAM with the same structure.

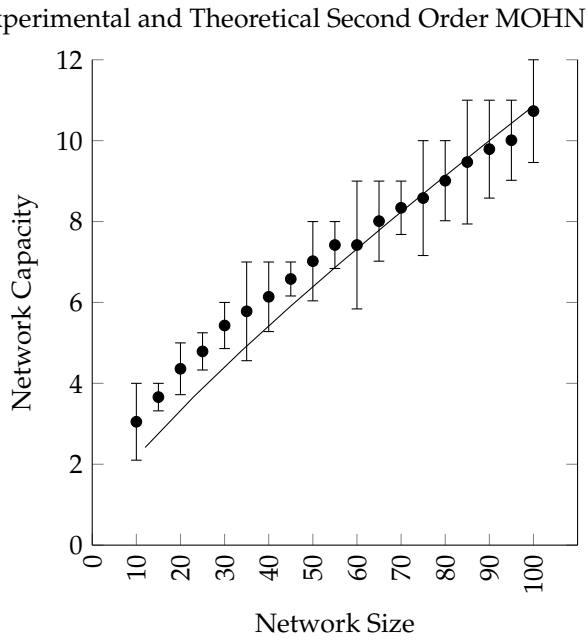

Figure 4.19: The mean and inter-quartile range of the capacity of second order MOHN networks of varying sizes trained with weighted Hebbian learning and the theoretic capacity of similar HNNs trained with simple Hebbian learning (single line).

4.5.5 *Linkage Order and Network Capacity*

The attractors of a CAM can be discovered from a function that maps patterns to their distance from the closest local maximum using the weighted Hebbian learning rule. However, weighted Hebbian learning does not attempt to minimise MSE between the MOHN output and the distance function. This section investigates some properties of the Hamming similarity based function compared to the energy function learned by the weighted Hebbian approach.

4.5.5.1 *Experimental Setup*

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. Once the network has reached capacity, the set of patterns that it has learned are used as the local optima in a multiple pyramid function as in equation 4.4. 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.

4.5.5.2 *Results*

Figure 4.20 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. The Hamming similarity function behind these attractors undergoes a Walsh decomposition and the highest non-zero order of the resulting coefficients is recorded. This highest order is counted across many trials for representation in the histograms. As capacity grows, so does the highest order of non-zero Walsh coefficients. It is clear from the histograms that many functions may have their attractors represented by a second order Hopfield network, even if the underlying structure of the function is of higher order. This is of particular interest if the reason for modelling the function is to find optima as part of a heuristic search.

The cost of using a low order model (second order alone in this case) is high model bias and the presence of spurious attractors (more on this in section 4.6.1). The benefit is that there is no need to learn the full function in detail or to match its complex structure if all that is needed is a CAM with attractors in the right place.

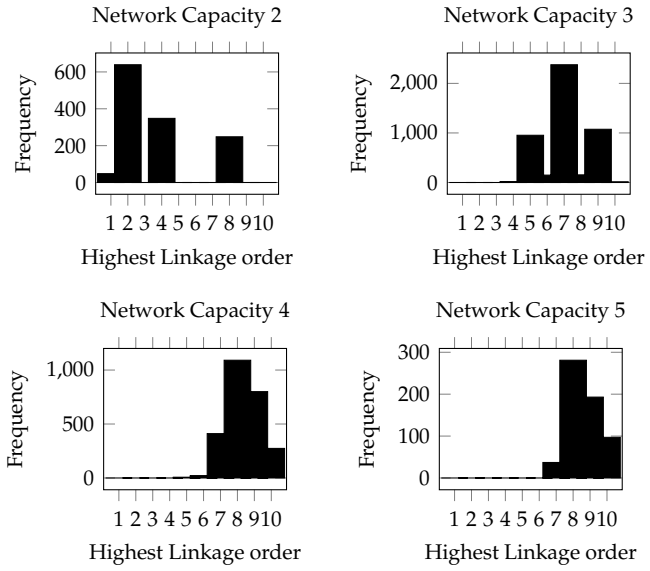

Figure 4.20: Histograms showing the frequency of the highest linkage order across 10,000 trials, organised by Hopfield network capacity. Networks are trained with the standard Hebbian rule. Networks with capacity greater than 5 require a number of units greater than that for which it is practical to run multiple Walsh decompositions.

4.6 Constraint Learning

Constraint learning is akin to optimisation in that it is a process designed to generate patterns that are very good examples of something based only on the scores attributed to a set of examples. Often the scored examples that are available do not score very highly, and certainly don't need to contain perfect examples. Rather than producing one optimal example, however, the goal is to produce many good examples. The desirable patterns should occupy multiple local minima in the fitness function.

4.6.0.3 *Experimental Setup*

An experimental example will clarify. In this experiment, the constraints to be learned are those that define symmetry, defined in equation 4.1. A second order MOHN was built using the weighted Hebbian learning rule of equation 3.4 with no error descent or structure discovery. For this task, the absolute value of the MOHN output need not be accurate as long as the attractors are correct. Training data was generated at random, scored with the fitness function of equation 4.1 and used to train the MOHN. At regular intervals during training, the MOHN was searched by picking random start points and settling to an attractor state, which was then scored. The maximum output of the symmetry function is one. When the number of different start points that all lead to attractors that score one, the process is terminated.

Figure 4.21 shows some examples of start points and their associated attractors after training a second order MOHN on random samples scored using equation 4.1. Patterns were generated in a 6 x 6 image of binary pixels. There are 2^{36} (68,719,476,736) possible patterns in such a matrix. Of those, there will be one vertically symmetrical pattern for every possible pattern in one half of the image. There are 2^{18} (262,144) such half patterns, representing 0.00038% of the total number of possible patterns. The network was allowed to learn until ten different symmetrical patterns had been found. At this point, the learning process was terminated and the network was tested with a set of local searches designed to count the number of attractor states learned.

4.6.0.4 *Results*

Across 50 trials, the 36 node MOHN took an average of 9932 pattern evaluations before it terminated having found 10 perfect scoring patterns. A record of the samples made showed that none of the randomly generated patterns used during training gained a perfect score, so the network was only trained on less than perfect patterns. The average trained model's capacity was found to be 132 patterns, which gives an average of 75 fitness function evaluations per attractor found. A MOHN with 36 inputs has 1260 second order weights, so over 9000 fitness evaluations is higher than would be needed with regression learning, suggesting that the weighted Hebbian approach is not efficient.

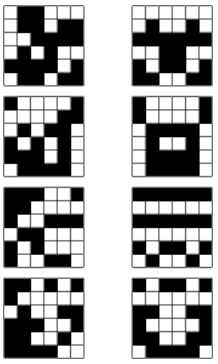

Figure 4.21: Random start points and their associated attractors in a MOHN trained using a fitness function that measures vertical symmetry.

A similar example involves learning the constraint of 'horizontal', which is defined as the number of variables in each row of a square image that are all the same value (all 1 or all -1). Figure 4.22 shows some examples of starting points and attractors after training on the fitness function of equation 4.10, which is

$$f(X) = \sum_{i=1}^{\sqrt{n}} \frac{e(i)}{\sqrt{n}} \tag{4.10}$$

where $\sqrt{n}$ is the number of rows in the square image and $e(i)$ returns the largest number of variables in row i that share the same value. As before, none of the training examples were a perfect example of a horizontally consistent pattern, but the constraints between pixels required to produce such a pattern were discovered by the learning algorithm based on the relative scores of a small set of suboptimal example patterns.

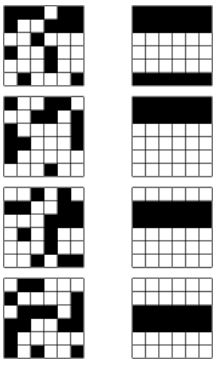

Figure 4.22: Random start points and their associated attractors in a MOHN trained using a fitness function that measures horizontal consistency.

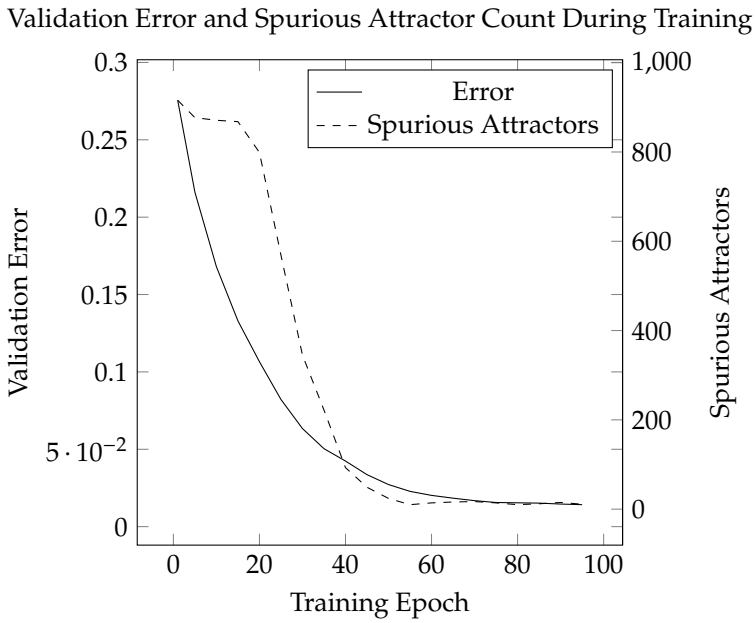

Figure 4.23: As the number of learning iterations increases, the validation error decreases as does the number of spurious attractors in the model.

4.6.1 *Energy Function Regression Learning*

The previous section addressed the task of learning a function in which local minima in a function underlying a data sample were also local minima in the MOHN's energy function. This section investigates the effect of learning that underlying function with more precision, fitting the energy function as a regression model of that which generated the data. Two experiments were carried out, the first measured the effect of SGD learning on the number of spurious attractors in a MOHN and the second measured the effect of using SGD on MOHN capacity.

4.6.1.1 *Spurious Attractor Removal*

Experiments were run in which a 100 neuron MOHN was trained on a function that contained four true attractor states. During training, the RMSE was recorded and the number of spurious attractors in the MOHN was estimated by picking uniformly random starting patterns and settling to an attractor point. This was repeated until 50 consecutive starting points were found not to add a new attractor to the set. Figure 4.23 shows the average results of running 100 trials on different randomly produced data sets of 20,000 samples from the Hamming similarity based function of equation 4.4. The number of spurious attractors drops with the RMSE. The first iteration of the learning process was a weighted Hebbian step in which all four attractors were found, so the target attractors were always present in the network from iteration 1.

More generally, the reduction in the number of spurious attractors depends on the ability of the structure of the network to represent the underlying function in which the only turning

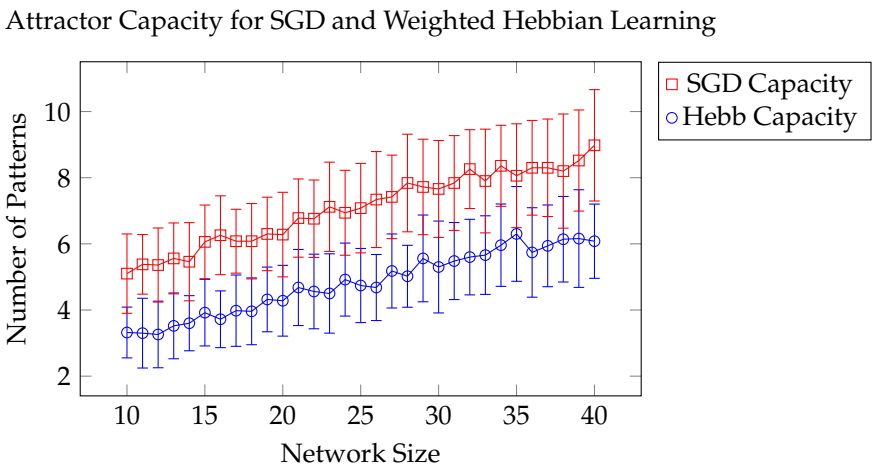

Figure 4.24: The mean and standard deviation of the capacity of a fully connected Hopfield Network trained with the Hebb rule and stochastic gradient descent.

points are the desired attractor patterns. In principle, a MOHN can represent any function (though time and memory constraints may prevent it in practise) to arbitrary accuracy, so for any set of target memories, there exists a MOHN that can not only represent those memories as local optima, but that also contains no spurious attractors.

4.6.1.2 *Improving Capacity*

Additionally, SGD allows a fixed size MOHN to increase its capacity over the equivalent trained with the Hebbian rule. The network capacity experiments described above were repeated with networks from size 10 to 40 with each trial learning one MOHN with the Hebbian rule and another with SGD. In each trial, a set of target patterns were generated at random, one at a time. Each pattern was learned by one MOHN using the Hebbian weight update rule and by another MOHN by a process of adding the pattern to the target set of a Hamming similarity based function and re-learning the pattern set from scratch using SGD. Each MOHN learned the same set of patterns until one failed, at which point the other continued to add patterns until it too failed. The number of patterns stored in each was recorded.

Figure 4.24 shows the average and standard deviation of the capacity of MOHNs from size 10 to 40, calculated from 50 trials of each learning rule at each network size. The learning process for SGD is longer and requires all of the previous memories to be present each time a new memory is added, but it has the advantage of providing a larger capacity and fewer spurious attractors.

4.6.2 *Visualising Network Structure*

The MSDA produces useful summary statistics during learning. As the structure evolves, the weight profile and the weight probability distributions may be reported and analysed to understand the progress being made. This section illustrates the structure discovery process using the same k-bit trap function described above. A visual representation of network structure is used to produce an image with n columns and j rows where each column represents a neuron and each row represents a single weight. The pixel at coordinate (i, j) is plotted if W_j is connected to X_i and its colour reflects the strength of the connection. If W_j is not connected to X_i, then no pixel is plotted. The weights are sorted in combinatoric order, with first order weights at the top of the image, second order weights below them, and so on. If a weight is not present in the network, it does not appear in the image, so the height of the image depends on the number of weights in the network.

Figure 4.25 shows an example for a correctly fitted 5-bit trap problem over six repeated traps. The interactions among the neurons in each trap are plain to see, as is the lack of inter-trap connections. Images such as figure 4.25 provide an insight into the function that has been learned and the complexity of the representation of that function. They also allow for a human led phase of learning. If a small number of weights were missing from figure 4.25, it would be easy for the human eye to spot them and add them manually to the model for a final round of learning.

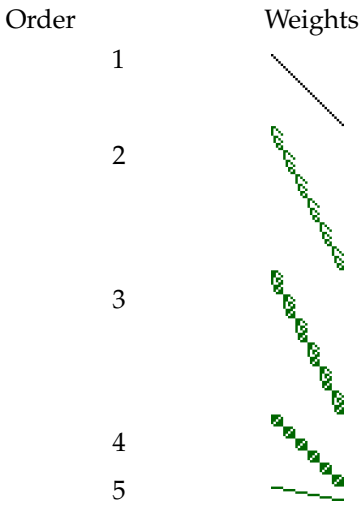

Figure 4.25: The weight structure of a MOHN after learning a 5-bit trap problem using MSDA. The groupings of the weights show how the trap function is made up of the sum of six independent functions concatenated across the inputs. Each function acts on a non-overlapping subset of the inputs and is fully connected within that set.

The structure of a network may be monitored during training both by full network representations such as that in figure 4.25 and by summary information about weight orders and neuron contributions. Figure 4.26 shows the structure of a network at selected points during the structure discovery of a 5-bit trap function. Weights at each order are easily identifiable. The behaviour of the algorithm is exposed as the network first grows (partly due to a higher critical value for the t-test) and then shrinks to the correct structure. It is also clear that the network has cleared the insignificant weights away from the lower orders before the higher order weights. Monitoring the image of a network allows the user to understand the current solution's level of complexity and the rate at which it is changing, allowing decisions to be made about terminating the process or altering the structure by hand.

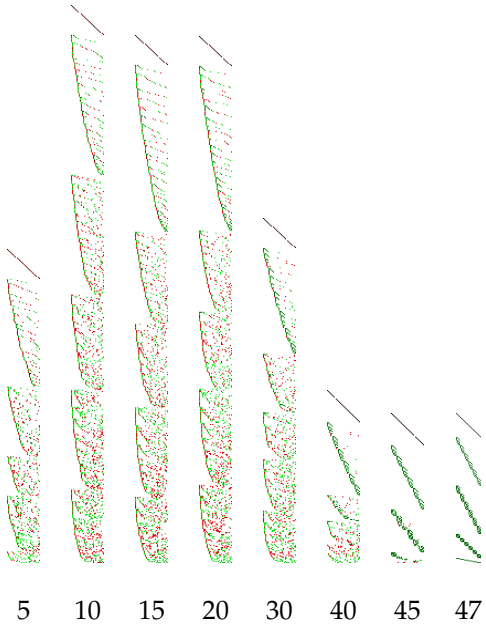

Figure 4.26: The structure of a MOHN during MSDA as a 5-bit trap problem is learned. The number below each column indicates the number of iterations of MSDA at which the snapshot was taken. Positive weights are green and negative are red.

Figure 4.27 shows the weight counts for each order during training for the 5-bit trap function. No upper limit was imposed on the order of weights added, but the Laplace distribution forced them to zero after order 6. By monitoring the number of weights used as training progresses, the user is able to gain an insight into the size of the remaining search space and the weight orders that remain to be explored. This helps the user make decisions about when to stop training and allows some insight into the likely quality of a model from their data. The MOHN can provide additional metrics such as the number of weights tried since a new significant weight was added, which provides further insight into whether continued training is likely to yield improvements in error.

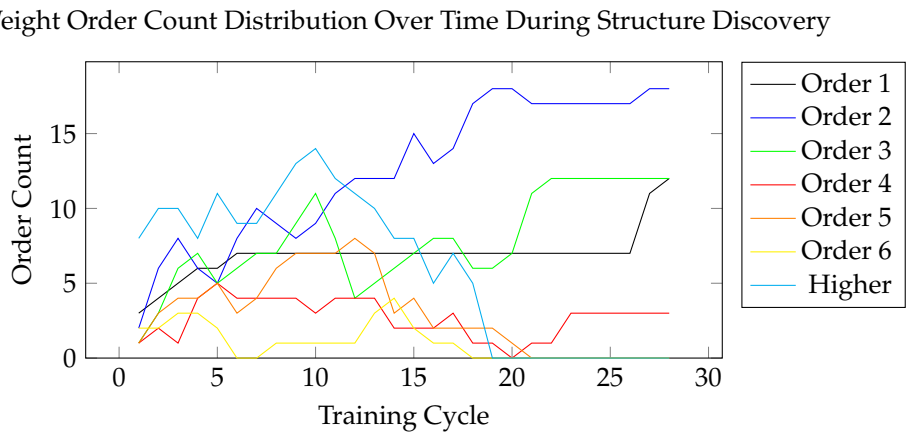

Figure 4.27: Weight counts at each order during MSDA for a 5-bit trap problem. Each line represents a weight order and shows the number of weights of that order the network contained at each iteration of MSDA. The far right hand points show the correct configuration.

4.7 Network Search Experiments

Section 2.2 described a number of ways of searching for attractor states in a MOHN. They were repeated network settling (random restart hill climb, RRHC), local search with high order kicks (iterated local search, ILS), high order search guided by weights (weight satisfaction search, WSS), search by removal of attractors (local optima suppression search, LOSS) and simulated annealing (SA). This section compares their performance.

4.7.1 *Hamming Based Functions*

The first experiment aimed to find the single global optimum from a 100 neuron network that was trained on samples from a fitness function with ten randomly placed attractors. The function output was from zero to one, with a single pattern producing the global maximum of one and nine other randomly placed patterns producing a local maximum of 0.9. All other input patterns produced values less than 0.9. The MOHNs were trained using the weighted Hebbian rule with no error descent, so learning was fast, but the number of spurious attractors was high, making the task of finding the global optimum more difficult. In fact, there were on average 148 spurious attractors in each network tested in addition to the local optima that were loaded as part of the test.

1000 trials were run, each with a different randomly produced fitness function. The five different search algorithms were used to search for the global maximum in each network and the number of times the algorithm needed to restart after a local optimum was visited was recorded for each. A restart for SA occurs after a full pass through its cooling schedule, meaning that a single pass of SA is generally much longer than a single hill climb of any of the other

algorithms. Table 4.6 shows the average number of restarts needed for each method. A t-test comparing the LOSS count with each of the others was significant at $p = 10^{-7}$.

Method	RRHC	ILS	WSS	SA	LOSS
Restarts	18	15	16	257	12

Table 4.6: The average number of restarts needed to find the global maximum across 1000 trials of randomly generated functions. On average, functions contained 148 local maxima.

4.7.2 *K-Bit Trap Functions*

Finding the global optimum for a k-bit trap function presents a greater challenge for algorithms that make single step improvements (such as the hill climbing part of those described here) as the score of a pattern is improved by setting a variable to -1 in all cases except those where the other $k-1$ of the k bits in a set are already equal to 1. Consequently, most hill climbs will take the network to a state where all the neurons have a value of -1. In this situation, the LOSS algorithm removes the all negative state as an attractor and subsequent attractors begin to include values of 1. This allows the algorithm a better chance of moving to the target state of a network where every neuron value is 1.

These experiments used a 10 variable function with a trap size of five, so the function contained two separate traps. The function was learned by a MOHN with full connections at all orders from one to five, so it was able to learn the function perfectly from samples from the fitness function. The resulting MOHN was searched using RRHC, WSS and LOSS and the results compared.

Table 4.7 shows the results of running 100 separate trials where a 5-bit trap function was learned from fitness function samples and the resulting network was searched using WSS, LOSS and RRHC. In these experiments, simulated annealing was quickly found to be very poor and was not included in the analysis. The only way for the RRHC algorithm to find the global optimum is for it to start very close to the solution, with each block (each k bit trap) being either optimal (all values equalling one) or only one step away (k-1 values equalling one). WSS climbed directly to the global maximum without any restarts. This particular function was made up of only two concatenated traps, so the search involved a full enumeration of each separate sub network. In cases where the structure reveals a large number of small sub networks, WSS is an obvious choice for finding a global maximum quickly. In reality, however, few functions will be quite so neat in their structure.

RRHC Restarts	LOSS Restarts	P-value
211	129	0.0003

Table 4.7: The average number of restarts made when searching for a single global optimum in a MOHN trained on a 5-bit trap function.

This section has presented some early evidence that LOSS is able to reduce the number of restarts required when searching certain types of function. It is most likely that other, different functions will not be amenable to this type of search. Search and fitness modelling will be considered in more detail when the MOHN approach is compared to EDAs in section 4.11.

4.8 Measuring Function Complexity

This section addresses the following question: *Given two different functions, each implemented by a MOHN, is it possible to consistently compare them in terms of their complexity?* The complexity of a model may be defined in terms of either its structure, its behaviour or its performance. Structural complexity is reflected by the number, size and (for higher order models like MOHNs) the order of the parameters that define them. Behavioural complexity is defined in terms of qualities of the output of the function such as its smoothness, sensitivity and the number of turning points it possesses. Performance complexity is equivalent to model bias and can be defined in terms of accuracy on a test set or cross validation run.

There are many possible measures of function complexity. Here three simple measures of network structural complexity are proposed: the number of weights, the sum of the weight orders and the L_1 norm of the weights. The number of weights in a network is simply the size of the weight set, $|\mathbf{W}|$. The sum of the weight orders is calculated as

$$\sum_{j=0}^{|\mathbf{W}|} |\mathbf{I}_j| \tag{4.11}$$

where $|\mathbf{I}_j|$ is the size of the set of neurons connected to weight W_j. The L_1 norm of the weights is

$$\sum_{j=0}^{|\mathbf{W}|} |\omega_j| \tag{4.12}$$

where $|\omega_j|$ is the absolute value of the weight value ω_j.

4.8.0.1 *Experimental Setup*

1000 MOHNs with 10 inputs connected by 10 weights were created by sampling weights of uniformly random order and with uniformly random weight values in $[-10, 10]$. From each

resulting MOHN, measurements were taken of the average weight order, the L_1 norm of the weights, the number of attractors and the average change in output produced by flipping a single random input bit, taken over 500 random patterns (sensitivity). The correlation of each pair of measurements was calculated and is shown in table 4.8. The highest correlation is between the number of attractors found and the average order of the weights in the network.

	L_1 Weights	Sensitivity	Attractors
Av. Order	-0.03	0.83	0.94
L_1 Weights		0.47	-0.03
Sensitivity			0.77

Table 4.8: Correlations between different measures of MOHN complexity.

4.8.1 *Complexity and Training Example Requirements*

The number of training data points required to build a statistical model depends on how noisy the data is and how many parameters there are in the model. A linear model with j parameters can be trained with a minimum of j unique noise free training points. This experiment demonstrates that, as long as the correct structure is known, this limit holds for networks with weights of any order.

4.8.1.1 *Experimental Setup*

100 functions were built by constructing MOHNs with weights of uniformly random values at uniformly random orders. The structure, but not the weight values, of each MOHN was copied onto a new MOHN. A sample of unique uniformly random input patterns was generated and evaluated using the energy function of the source MOHN. The size of this sample was equal to the number of weights in the MOHN. OLS was used to train the new MOHN on the training data and the new MOHN was then tested on another random data set of equal size. Each MOHN was 100 neurons in size and had 1000 randomly placed weights. The correlation between the output of the trained MOHN and the target MOHN was measured for each trial.

4.8.1.2 *Results*

For every trial, regardless of the order of the weights in the network, the MOHN with the correct structure was able to learn the weight values and produce a correlation on validation data between the MOHN output and the target output of 1. This is not an unexpected result, but it demonstrates that the order of the weights makes no difference to the sample size required to learn their correct values, once the correct MOHN structure has been learned. A MOHN with

the correct structure and j weights needs j random noise free samples of $f(X)$ to learn $f(X)$ perfectly.

These two experiments lead us to two conclusions of note. Firstly, that the number of attractors a MOHN contains is highly correlated with the average order of the weights it contains and secondly that the number of training samples required to learn a function is not dependent on the number of attractors or the order of the connections. It is purely dependent on the number of parameters in the model.

It is worth highlighting the fact that the number of data points required only equals the number of parameters in the model when the model is correct. Generally the correct structure of the model is unknown, making the number of parameters and, consequently, the required sample size also unknown. Further more, if the model structure is wrong, and the sample size equals the number of parameters, then the training error will be very low. In such cases, increasing the training set size or (similarly) using an independent validation set provides valuable information about how well the model fits the data. An experiment illustrates the point.

4.8.1.3 *Experimental Setup*

A first order only model with 30 inputs was trained on random pyramid functions with varying numbers of local optima. The greater the number of optima, the more complex the underlying model that is required to model them. For each trial, a random pyramid function with k local optima was generated and the first order model was trained on samples of increasing size, m, from 30 (just sufficient to learn the data) up to 276. For each training sample size, the correlation between the training data and the model output was calculated, as was the correlation between the test data and the model output. Fifty trials were repeated for each k and the correlations averaged.

4.8.1.4 *Results*

When $k = 1$, the function has a single local maximum and is learned perfectly by the MOHN in 30 samples, producing zero error on test data. For $k > 1$ the training correlation is 1 when $m = 30$ and drops as m grows larger. Conversely, the test correlation starts low when $m = 30$ and grows, but never reaches the training correlation level. Figure 4.28 shows the results for $k \in \{4, 6, 10, 15, 50\}$. These experiments involved noise free data and in such cases (as is common when modelling a fitness function model), the sample size should allow for the number of parameters in the model plus a number to perform an independent test. As the quality of the model structure improves, training and test correlation converge.

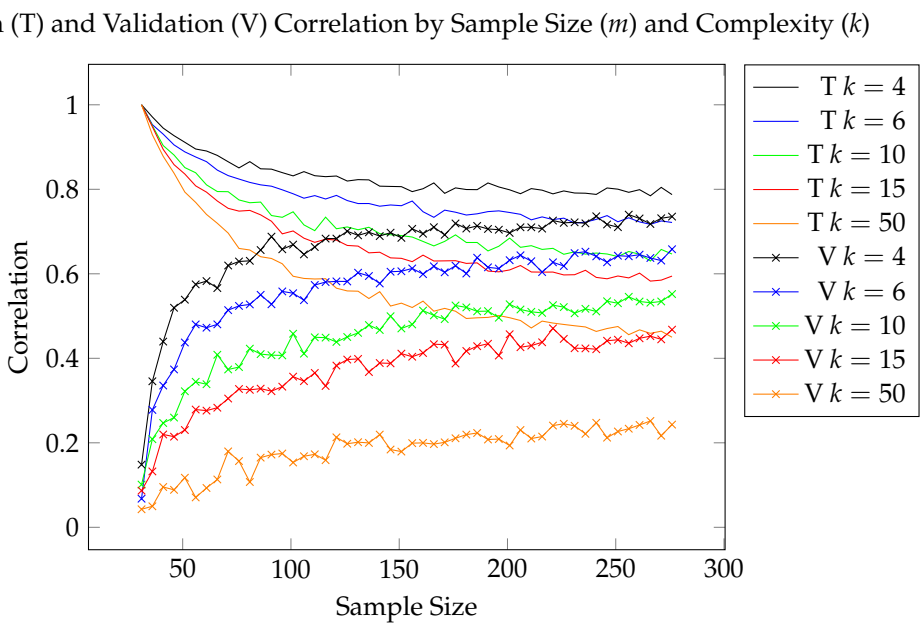

Figure 4.28: The training and test correlation between a model and the target data as sample size grows when the model is under fit, plotted for models of different complexity. k is the number of local maxima in the target function, which is used as the complexity measure.

4.8.2 *Conclusion*

That concludes the experiments on small problems. They have provided some insight into the use of a MOHN, but not really tested it in earnest. The next sections test a MOHN on real data and compare it to other methods in the recent literature.

4.9 Consumer Profile Data

This section describes a number of experiments using real data describing consumer behaviour. The data makes use of Experian's *Enhance* data set, which is a demographic data appending service with a mixture of personal, household and postcode level variables. Personal level variables include gender, age band, marital status and occupation. Household level data includes income, household composition and length of residency. Experian data is used to target marketing material, design advertising campaigns and literature, and make decisions about consumer credit. The data used in the experiments reported here are derived from real data from a real client.

Experian's Enhance variables are all coded as discrete variables. Numeric variables such as age and income are split into bands so all of the variables lend themselves to a 1-of-k binary coding where k is the number of values a variable can take (its cardinality). This creates a MOHN with a number of nodes equal to the sum of the number of values each variable can take.

Formally let V be a vector of m nominal variables, V_k where $k = 1 \ldots m$ each of cardinality C_k. These are mapped onto n binary variables, $X = X_1 \ldots X_n$ where $n = \sum_k C_k$. Contiguous non-overlapping subvectors in X of size C_k represent the possible values V_k can take. The subvector of X corresponding to variable V_k is X^k and X^k_l represents the l^{th} value of variable V_k. The values of X are constrained so that only one element in each X^k can take the value of 1, all others being set to -1.

4.10 Clothing Mail Order Case Study

Data describing the demographics and annual spend of 14,609 customers of a mail order clothing retailer were used in this case study. The goal was to build a model capable of taking the description of new potential customers and providing a prediction of their expected annual spend. Demographic data was appended using Experian's Enhance data set and a subset of variables describing gender, age, marital status, income, number of children and home ownership status was chosen for modelling. The 1-of-C_k coding across these variables resulted in a MOHN with 36 nodes.

4.10.1 *Model Training*

The data set was initially split into a training set containing 70% of the data and a test set containing the rest. The training data was further divided into training and validation (or multiple cross validation) sets. When a single validation set was used, it contained 30% of the

original training split, leaving 70% to be used to learn the weight values. The splits were made by selecting training examples uniformly at random without replacement. For cross validation, 10 folds were used.

There are a number of hyperparameters that can be tuned when using the MSDA and a subset of the available combinations were explored using a grid search for those that produced a promisingly low error when trained on the training data and tested on a 30% validation split. The grid search parameter sets were:

Hyperparameter	Grid Set
Learning Method	{SGD, Lasso}
Initial weights	{100,500,1000,4000,6000}
Added weights	{100,200,500}
Iterations	{10,30,50,80}
For SGD	
Learning rate	{0.1,0.3,0.6}
Critical p-value	{0.9,0.6,0.3,0.1}
For Lasso	
λ constraint	{1,2,3,5,7}

During lasso learning, a number of different values for λ are used to produce a number of different models, each with a level of regularisation determined by its λ. In these experiments, 7 models were built (i.e 7 different λ values were set) and the model used to choose which weights to remove was selected according to its place in the list of regularised models, where model 1 has the least regularisation and 7 has the most. The actual value of λ depends on the residual correlations between features and the output so their relative indexes are used to specify the grid search. The settings found to give the best generalisation (the lowest RMSE on the test data) were:

Hyperparameter	Value
Learning Method	Lasso
Initial weights	4000
Added weights	500
Iterations	30
λ constraint	3

The training data partition was then split further for use in a cross-validation training process where ten validation sets are created, each consisting of a unique 10% of the training data such that each data point is included in exactly one validation set. Each of the ten training sets

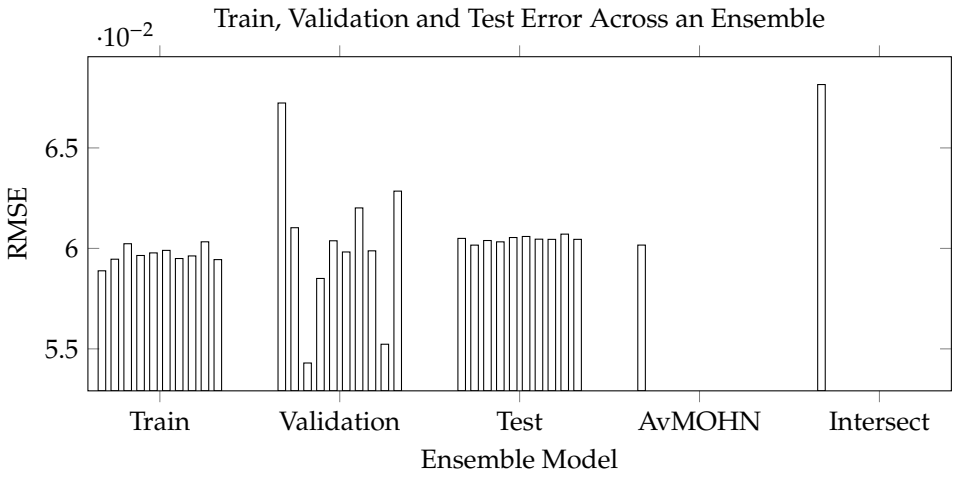

Figure 4.29: RMSE measures across 10 validation folds for training, validation and test data sets and of the average MOHN on the test set. Note that the Y axis does not start at zero, which makes the differences easier to see.

consist of the data that is not set aside for their corresponding validation set. Ten models were then built, one from each training set. The models were tested on their own training data, their own validation data and on the independent test partition.

The ten models were trained using the MSDA described in section 3.3 with the hyperparameters described above. The networks in the ensemble were combined using both the intersection and averaging methods described in section 3.6.1.1, giving 12 MOHNs in total.

4.10.2 *Results*

Figure 4.29 shows the root mean squared error for the training, validation and test data from each of the 10 validation folds of the experiments described above. Figures are shown for the accuracy of each model on its own training and validation split and when tested on the single separate test data. A single test error figure is also given for the average MOHN and the intersection of the ten ensemble models. There is variation in the performance of the ensemble members on their validation sets, but they all perform with very similar results on the test data. The average MOHN performs very slightly better (though not statistically significantly) on the test data.

Examination of the intersection model, shown in figure 4.31, can also reveal some clues as to the correct structure to explore. The intersection model only contains first and second order weights, suggesting that those orders are consistently important in this case. It may be the case that second order weights are sufficient, but that the intersection of the models does not contain all those that are required. This is investigated next.

4.10.3 *Further Pruning*

The average MOHN is quite large. The obvious question to ask is whether it is possible to prune the networks further without appreciable loss of performance. When a network is pruned, the remaining weights need to be re-learned so an iterative approach to pruning and testing can be carried out by removing a subset of weights and then retraining and testing. At each iteration, weights may be removed according to a criteria of increasing severity. In this example, each iteration removes weights of the highest order currently in the model.

At each iteration, the highest order weights were removed (starting at five, which was the highest order in the original network) and the network was re-trained on its training data and tested on its validation split. Table 4.9 shows the results, where it can be seen that the test error actually improves slightly for the first and second order network, although the training error rises. This suggests that the network with higher order weights had overfit, leading us to adopt the second order model.

Highest order	Weights	Train RMSE	Test RMSE
5	5067	0.0585	0.0647
4	4557	0.0588	0.0632
3	3134	0.0588	0.0633
2	660	0.0604	0.0633
1	36	0.0707	0.072

Table 4.9: Test and train error after removing weights of successively lower order and the size of the resulting network. The final row is a standard first order linear regression. The error differences look small, but make a significant difference to prediction accuracy.

4.10.4 *Gaining Knowledge from the Network*

Having built and tested the MOHN, there are two ways in which knowledge may be extracted, in addition to the usual activity of making predictions for new potential customers. Firstly, the attractors of the model, which represent local maxima in amount spent, can be analysed and then the weights of the network can be studied.

4.10.4.1 *Extracting Local Maxima in the Spend Function*

By allowing the network to settle from different starting points, profiles describing high spending customers may be extracted from the network. The 1-of-C_k coding scheme for inputs means that unless precautions are taken against it, an attractor can occupy a point where any number of the values for each variable are set to 1. This can lead to nonsensical answers where

setting many of the possible values of a single variable can inflate the predicted output. If the one bit only rule is not enforced, maxima in the function will be points where every value that has a positive effect is set to one. A MOHN can represent a 1 of C coding across a set of nodes but this requires full connection among all the nodes at all orders up to C, giving 2^C weights in the group.

This inefficiency can be avoided by altering the network settling algorithm to ensure that it settles only on patterns where exactly one of the neurons in each variable's subset takes a value of 1. This was implemented by repeatedly finding the single neuron in each variable's subvector (i.e. the single value from those that the variable could possibly take) that makes the highest contribution at the current point. Simple hill climbing takes the values across the network to a point where the predicted output is locally maximised. This process is repeated from random starting points to generate a distribution of optima with associated outputs and counts. Algorithm 18 describes the process.

Algorithm 18 Settling a trained MOHN to an attractor point across nominal variables

repeat
  $ch = FALSE$ ▹ Keep track of whether or not a change has been made
  $visited = \emptyset$ ▹ Keep track of which variable have been set
  repeat
    $i = rand(k : k \notin visited)$ ▹ Pick a random variable
    $cur = f(X)$ ▹ Make a note of the current predicted output for later comparison
    $Maximise(X^k)$ ▹ Set the single neuron, X_i in the subvector X^k that maximises the output of the network when $X_i = 1$
    if $f(X) \neq cur$ **then**
      $ch = TRUE$
    end if ▹ If a change was made to the neuron's output, note the fact
    $visited = visited \cup k$ ▹ Add the variable's index to the visited set
  until $|visited| = m$ ▹ Loop until all variables have been updated
until $!ch$ ▹ Loop if any neuron value has changed

Algorithm 18 was run from 200 random starting points on the MOHN that was trained on the data described above and seven different local maxima were found. Figure 4.30 shows the predicted output (i.e. the customer's predicted spend) from the model at each attractor point, plotted against how often that pattern was found from 200 random starting points.

The most common pattern described well off married women with children who own their own home, which matches the client's known profile very well. One of the less frequently found (only three times in the 200 trials) attractors revealed a pattern of young unmarried women with no children, which might indicate a market that is currently under exploited for the company.

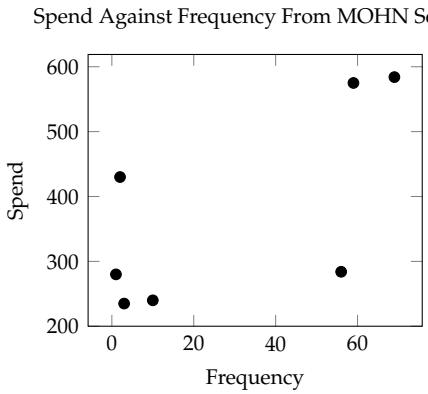

Figure 4.30: The seven attractor points of the customer profile optimisation search plotted as predicted spend against the number of times the attractor was found.

4.10.4.2 *Studying the Weights*

Although the intersection MOHN was not the most accurate, its output still had a correlation with the actual sales data in the test set of 0.82. It has the analytical advantages of representing a set of weights that all ten MOHNs in the ensemble discovered to be significant while being small enough to visualise and study. The network studied in this example is the result of finding the intersecting weights across all ten of the cross-validation networks and then re-training the resulting small network on all of the training data so that the weight values are coherent with the structure.

Figure 4.31 shows a visual representation of the weights of the ensemble intersection, annotated to show the variables that the nodes represent. The first order connections show positive contributions to spend from being married, from an upper-middle age band, from people with higher incomes and from people with children. The green squares indicate a positive influence on spend and red indicates a negative influence. Looking at the second order connections reveals some exceptions to those patterns, however. For example, it can be seen that young (the second column in age band, see rows 18 and 19 of the second order weights) customers with no children and young customers with high income both spend a lot with the company. This observation agrees with the findings from the analysis of attractors described in the previous section. There are a few connections between pairs of values of the same variable (a good example is at the top of the married column in the second order weights). These connections have negative values, indicating the fact that they cannot both take a value of 1 at the same time.

4.10.5 *Comparing Ensemble Members*

One advantage of the explicit structure of a MOHN is that one can be compared with another in terms of structure. A measure of the difference between two MOHN weight sets, $\mathbf{W}$ and $\mathbf{V}$ is

the difference between the size of their intersection and their union as a proportion of the size of their union.

$$d = \frac{|\mathbf{V} \cup \mathbf{W}| - |\mathbf{V} \cap \mathbf{W}|}{|\mathbf{V} \cup \mathbf{W}|} \tag{4.13}$$

which measures the number of weights that appear in only one network as a proportion of the number of weights in both. Table 4.10 shows the proportional distance between network pairs in the ensemble where weight orders were limited to below five and table 4.11 shows the figures for the ensemble of second order networks. The second order networks are in closer agreement than those with higher order weights, showing less structural variance.

	1	2	3	4	5	6	7	8	9
2	0.74								
3	0.76	0.74							
4	0.73	0.73	0.76						
5	0.76	0.74	0.74	0.73					
6	0.75	0.75	0.76	0.73	0.74				
7	0.74	0.71	0.73	0.72	0.73	0.73			
8	0.76	0.75	0.75	0.73	0.75	0.76	0.74		
9	0.73	0.72	0.75	0.72	0.75	0.76	0.72	0.76	
10	0.72	0.71	0.70	0.71	0.72	0.73	0.70	0.73	0.72

Table 4.10: Proportional distance between each pair in an ensemble of ten order five limited MOHNs measured using equation 4.13

	1	2	3	4	5	6	7	8	9
2	0.40								
3	0.45	0.41							
4	0.40	0.39	0.43						
5	0.46	0.42	0.45	0.40					
6	0.44	0.42	0.46	0.42	0.44				
7	0.41	0.37	0.40	0.37	0.42	0.40			
8	0.44	0.43	0.41	0.37	0.43	0.45	0.41		
9	0.37	0.36	0.44	0.40	0.44	0.44	0.41	0.41	
10	0.39	0.38	0.39	0.36	0.40	0.43	0.38	0.39	0.38

Table 4.11: Proportional distance between each pair in an ensemble of ten second order limited MOHNs measured using equation 4.13

4.10.6 *Comparing a Multi Layer Perceptron*

For the purpose of comparison, the task of modelling the same data was performed using MLPs. The data sets used were the same as those employed to train the MOHN, with the same training, validation and test splits. The output values were rescaled to a range between zero and 1 across all of the data and the input values were left in $\{-1, 1\}$. Initially, for the purpose of performing a grid search for a promising set of hyperparameters for training the networks, the data were split into a training partition of 70% test partition of 30%. The grid search sets were:

Hyperparameter	Grid Set
η	0.1,0.2,0.4, 0.6, 0.8
Learning rate decay	1, {0.9,0.6,0.3} every {50,100,500} epochs
α	0.2,0.5,0.8
Hidden activation	Logistic, Tanh, ReLU
Output activation	Linear
Hidden Layers	{1,2,3}
Hidden units	{6,10,14,18,20}
Random weight range	{0.01,0.1,1}
Training epochs	20,50,100,500

A network with the following hyperparameters during learning performed the best:

Hyperparameter	Value
η	0.4
Learning rate decay	None
α	0.5
Hidden activation	Logistic
Output activation	Linear
Hidden Layers	3
Hidden units	4 per layer
Random weight range	0.1
Training epochs	100

Another 50 networks were trained with the same hyperparameters on the same training and test data to verify the variation due to random starting weights was smaller than the variation due to hyperparameter choice. The variance among the test error of this set (0.0000005) was much smaller than the variance across the grid search (0.007), indicating the choice of parameters was reliable.

The training partition of the original data set was then split into ten training sets, each consisting of 90% of the training partition. The remaining 10% in each case was used for validation. Ten MLPs were built, each trained on one of the ten training splits and validated on its corresponding validation split. Figure 4.32 shows the MOHN and MLP accuracy over the ten training and ten testing data splits.

Figures 4.33 and 4.34 show the predictions made by the average MOHN and the MLP ensemble, each plotted against the actual output from the data in a test set. The two plots appear similar and the agreement between the MLP and the MOHN can be seen in figure 4.35, where the MLP output is plotted against the MOHN output for the same data set.

Although the accuracy of the two models was similar, the advantages of using the MOHN were the insight into the relationships being modelled that the MOHN allowed, which also led to the ability to simplify the model, and the fact that the MOHN ensemble could be combined into a single model, rather than using all ten MLPs each time an averaged prediction is needed.

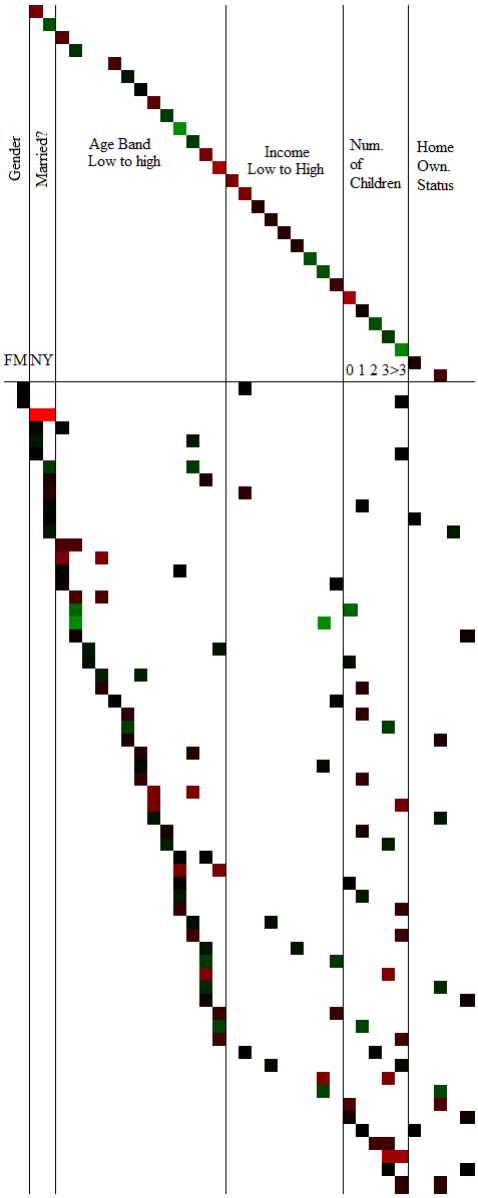

Figure 4.31: The weights of the intersection of all ten MOHNs in the clothing retailer example. Green indicates positive weights, red negative and the brighter the colour, the larger the size of the weight.

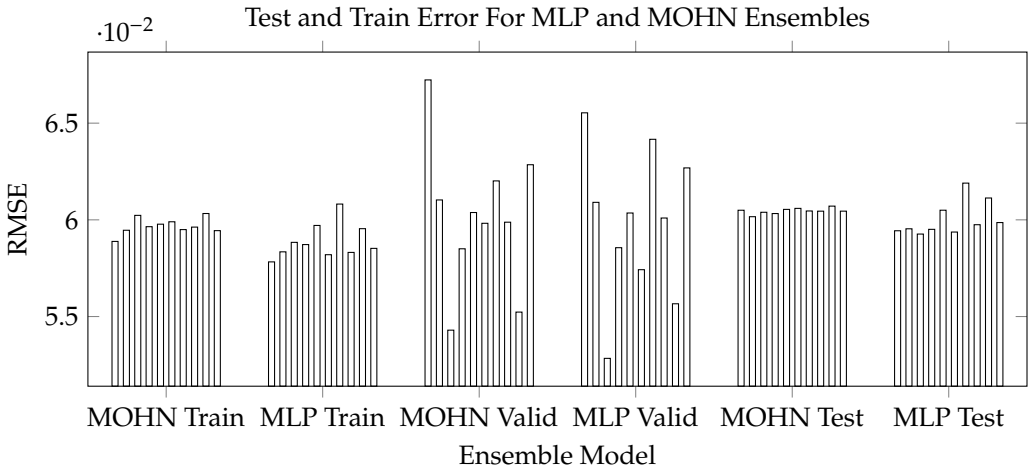

Figure 4.32: RMSE measures across 10 validation folds for training and test data for the average output of an MLP ensemble and an average MOHN model. Note that the Y axis does not start at zero, which makes the differences easier to see.

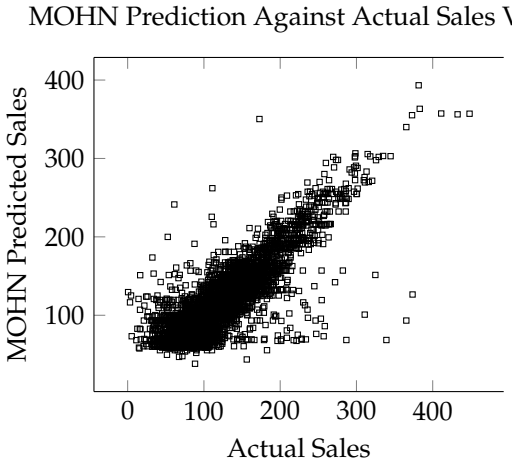

Figure 4.33: MOHN predicted output plotted against actual output from the data in a test set.

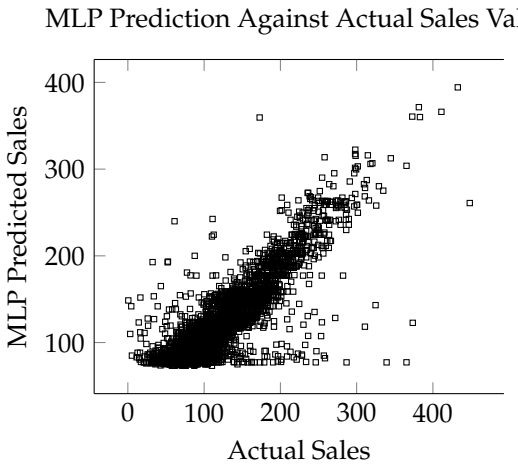

Figure 4.34: MLP predicted output plotted against actual output from the data in a test set.

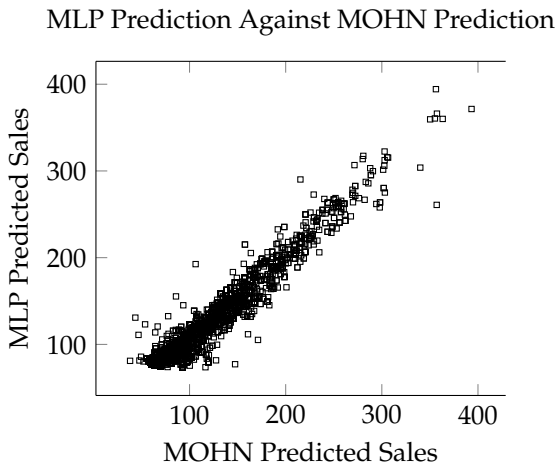

Figure 4.35: MLP predicted output plotted against MOHN predicted output from the data in a test set, showing the close agreement between the two.

4.11 Comparing MOHNs with EDAs

One way that a MOHN can be used as a surrogate fitness function to aid a heuristic search is to build an accurate model of the function to be searched and then search the MOHN, making use of its transparent structure. This is essential in situations where a sample of data rather than a fitness function is available, such as that described in section 4.9. It may also be useful if the fitness function is expensive to evaluate and the modelling process can complete in fewer evaluations than an alternative search heuristic. This may be particularly beneficial in situations where the same fitness function needs to be locally optimised many times from different starting points.

An alternative but related approach to building a surrogate fitness function is to build an EDA, which models the probability distribution of high scoring solutions and generates samples from that distribution. The EDA usually follows several generations of distribution modelling and sampling and attempts to introduce model bias so that the distribution models only part of the full distribution. This section addresses the question of whether it is more efficient in terms of fitness function evaluations to build a fitness function model (a MOHN) or an EDA. An EDA has one obvious advantage and one obvious disadvantage compared to a fitness function model (FFM). Its advantage is that the model it builds at each generation can be simpler than the full model required in a fitness function model as it only needs to represent those parts of the function that score well. Its disadvantage is that the quantity of data required to select the best from a population and model its distribution completely is more than that required to simply learn the fitness function. The question is whether or not a partial EDA can find the solution in fewer evaluations than a full FFM.

4.11.1 *General Experimental Methods*

The experiments that follow use MSDA to build MOHN fitness function models, which are then searched for optimal inputs with respect to the output of the function. The data are noise free samples from the target functions and the target for the training process is a validation error of zero. Earlier in this chapter, experiments with the MSDA under various hyperparameter regimes are described. The results of exploring hyperparameters in those experiments are applied to the experiments that follow. For all of the following experiments, most of the hyperparameters are fixed to a value that, while not optimal, has a high probability of allowing the algorithm to reach the target validation error (based on evidence from previous experiments). For example, the learning rate is fixed at 0.3 throughout. This was never found to be too high in previous experiments and although higher values may allow the algorithm to converge faster, that is not the primary goal.

The hyperparameters for the MSDA are described in section 3.3.10. They were set according to the following regime. When the lasso was used for parameter fitting, the level of regularisation is set to the minimal non-zero value chosen by the lasso fitting algorithm. When the parameters are estimated with SGD, there is always a parity learning step at the start, learning rate is always 0.3 and the number of epochs for a single iteration is fixed at 20. The p-value for the weight removal t-test begins at 0.5 and is reduced by 0.05 whenever an iteration of the algorithm fails to remove any weights, stopping at a minimum of 0.001.

The weight order distribution is always a discrete Laplace with mode=1 and $\lambda = 2$. The weight order update rule is always set to $\alpha = 0.6, \beta = 0.2$. The exploration/exploitation trade-off is set to full exploitation at all time. The algorithm ensures that there are never more weights in the MOHN than there are training data points. The schedule for emptying the used weight list is chosen by running the MSDA until it enters a phase where weights were being added and removed with no improvement in validation error. The number of iterations taken to reach that point is used as the reset schedule (usually the figure was rounded to the nearest 10 or 100).

The number of weights in the initial iteration and the number added at each subsequent iteration was set by default to be one third of the number of training examples. This was chosen to balance between exploring many weights at each iteration with the need to keep the number of weights below the number of training examples. Some experiments deviate from this regime, and where they do, the fact is noted and justified in the description of the experiment. Some of the following experiments compare the MOHN to an alternative EDA approach based on a paper from the literature. In those cases, the training set size is chosen to be smaller than that reported in the associated paper. Little effort is made to minimise sample sizes, as that is not part of the MSDA process, so the choices are a little arbitrary. From any chosen data set, 90% of the data is used for training and 10% for validation.

4.12 Comparing MOHNs and BMDA

The bivariate marginal distribution algorithm (BMDA) [103] is a second order EDA that models conditional probabilities with a pairwise variable interaction graph. Pelikan and Mühlenbein [103] present results that measure the number of fitness function evaluations required to find a global optimum for three different fitness functions using BMDA. This section considers one of them: the quadratic fitness function

$$f_q(X) = \sum_{i=1}^{\frac{n}{2}} f_2(X_{2i-1}, X_{2i}) \tag{4.14}$$

where $f_2(u, v)$ is

$$f_2(u, v) = 0.9 - 0.9(u + v) + 1.9uv \tag{4.15}$$

where $u, v \in \{0, 1\}$. For the sake of comparison, this domain is used in the following experiments and inputs of -1 to the MOHN are simply replaced with a value of 0 before evaluation with equation 4.15. Pelikan and Mühlenbein [103] pair variables from randomised locations, so there is no prior knowledge about which input interacts with which. Allowing the knowledge that the function has no interactions above second order means the search is among $n(n-1)/2$ second order and n first order interactions.

4.12.1 *Learning the Quadratic with the Lasso*

Armed with the knowledge that a second order function is sought, and the result shown in section 4.8.1 that with a noise free sample of data, the sample size should be no less than the number of parameters in the model, the minimum sample size required to learn the function may be calculated. Not knowing which pairs are significant requires them all to be considered, so a full first and second order model with n inputs would require $n(n-1)/2 + n + 1$ samples. The following experiment tests this limit on the quadratic equation described above.

4.12.1.1 *Experimental Setup*

A MOHN with first and second order connections was built and trained using the lasso with minimum regularisation and a sample of size $n(n-1)/2 + n + 1$ for n ranging from 10 to 120, the same as the range used in [103]. The correlation between the model output and the desired output of equation 4.14 was measured in each case. In this case, an additional validation set was generated, with size equal to 10% of the training data set size. The MOHN was then searched to find the optimal pattern, using weight satisfaction search.

4.12.1.2 *Results*

In every case for $n = 10 \ldots 120$, the model was able to learn the function to a correlation coefficient of 1 with the validation data and the weight satisfaction search moved directly to the correct optimum. Pelikan et al. [103] compare BMDA with a genetic algorithm with one-point cross over given the task of finding the global optimum of $f_q(X)$, which occurs when $X_i = 1$ in every element. They report that the average number of fitness evaluations that BMDA needed to find the optimum for $n = 10 \ldots 120$ were approximately double the $n(n-1)/2 + n + 1$ estimate made here. The GA took up to an order of magnitude more than that. In this case, the MOHN fitness function was able to model the function and find the optimum in around half the evaluations reported for BMDA and a tenth of those reported for the GA. The quadratic

also provides an example of a function that leads a first order hill climb to local optima, where any joined pair of variables both have values of zero. The structure revealed by the MOHN tells the weight satisfaction search which pairs are joined and so allows it to move from (0,0) across a pair directly to (1,1) without needing to try every possible pair. There are $n/2$ pairs in a function of n variables and the weights between them in the MOHN specify that they must have a positive product, so each weight defines two possible points. The full search requires the $n/2$ weights to try each of their two points, making the size of the search space equal n. To try every second order combination without the structural knowledge reveal by the MOHN would require $n(n-1)/2$ function evaluations. That number of evaluations were required to build the model, so any reduction in the sample size used to build the model represents an improvement over a blind second order search. This will be addressed in section 4.12.3.

4.12.2 *Reducing Evaluations Further*

Observing that the final, pruned MOHN after the lasso had forced the unnecessary weights to zero always contained $n + 1 + \frac{n}{2}$ weights, suggests that the number of evaluations could be reduced further. This is hindered by the fact that even though the majority of weights are not required, fitness evaluations are needed to discover which can be ignored. Regression algorithms such as the lasso and OLS cannot just ignore the unused weights. Brownlee [21], for example reports a experimental evidence that the quality of an OLS model suffers dramatically as the number of samples drops below the number of parameters being learned. To illustrate this point, an experiment was carried out to show the effect of adding unnecessary weights to a MOHN.

4.12.2.1 *Experimental Setup*

100 repeated trials of the following experiment were made and the results averaged. In each trial a single model of size n was built and pruned using the lasso with $n(n-1)/2 + n + 1$ random inputs evaluated with the quadratic of equation 4.14 and a fully connected first and second order MOHN. The resulting MOHN had its zero valued weights removed and was re-trained with a sample of size $|W|$ where $|W|$ is the number of weights it contained. New second order weights were then added one at a time to the model, connecting two nodes picked at random. The model was re-trained on the same smaller sample used in the previous step and the correlation between target and MOHN output recorded for both the training data and an independent test set. Two sets of experiments were conducted, one using OLS to learn the parameters and the other using stochastic gradient descent. In the SGD case the weights from the previous step were set to zero when each new weight was added, to ensure a fair comparison with OLS.

4.12.2.2 *Results*

Figure 4.36 shows the correlation between the MOHN output and the target from equation 4.14 for MOHNs of growing size trained using OLS and stochastic gradient descent. Every MOHN contained the correct weights, but an increasing number of unnecessary weights in addition. Using OLS, the correlation very quickly falls towards zero as new weights are added, as reported in [21]. Even though the model contains the correct weights, the presence of even a small number of unnecessary weights prevents it from learning if the sample is too small. The models trained using stochastic gradient descent maintain a high correlation between the training data and the target outputs for that data, but the correlation for data in the test set falls quickly. This highlights the need for care when building fitness function models as overfitting is still possible even if the data are noise free.

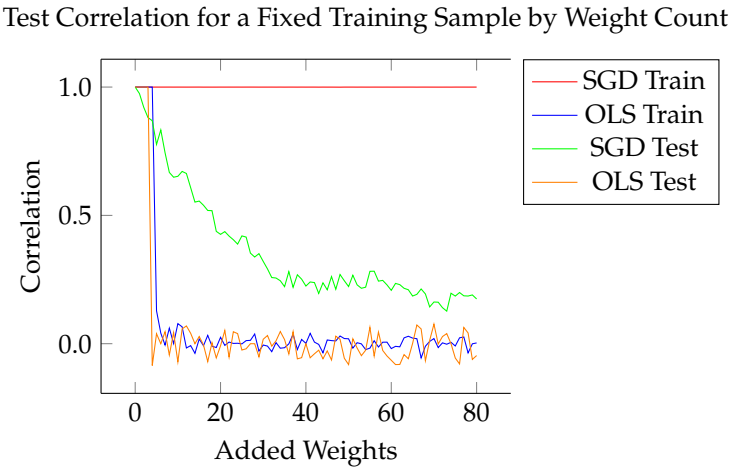

Figure 4.36: Train and test correlation for a fixed sample and a MOHN with incrementally added weights in addition to those needed to represent the function. OLS train and test correlation degrade at the same rate, SGD overfits, keeping the train correlation near 1 as the test correlation drops.

As long as the sample is larger than the number of weights in the model, it is possible to make some judgement about whether or not each weight is necessary. As the number of good weights increases, the judgement made about other weights improves. This suggests that it may still be possible to reduce the number of fitness evaluations by using the MOHN structure discovery algorithm. The next experiment tests this.

4.12.3 *Using Structure Discovery to Reduce Evaluations Further*

Although the lasso cannot be used when the number of weights exceeds the number of data samples, the knowledge that not all of the weights are going to be kept can be used to reduce the required sample size. In the following experiments, the size of the sample is kept smaller than

the theoretical lower bound for the network in question and the structure discovery algorithm is used to discover the right weights while maintaining a network that only ever contains a subset of the weights that are possible. The size of this subset of weights is managed to ensure it never exceeds the number of training data points.

4.12.3.1 *Experimental Setup*

The previous experiments based on the quadratic function used a fixed size sample to learn the fitness function model in the form of a MOHN. That sample's size equalled the number of weights in a fully connected first and second order MOHN. In these experiments, the training sample size was fixed at half that amount.

The MOHN structure discovery algorithm (MSDA) was run on the same range of problem sizes reported in [103], using the lasso for parameter estimation and weight removal. The hyperparameter settings described in section 4.11.1 were used, with the number of weights added at each iteration fixed at a third of the size of the training sample, up to a fixed limit to maintain a weight set smaller than the number of training examples. A maximum of 400 iterations of the algorithm were allowed with the used weight set being emptied every 50 iterations. Early stopping was permitted when the correlation between target and MOHN output was over 0.99. A final verification of the quality of the model was made by using weight satisfaction search to find the optimal input.

4.12.3.2 *Results*

The MOHN structure discovery algorithm (MSDA) was able to find the correct MOHN structure and weights in every trial from size 20 to 120, achieving a correlation over 0.99 and finding the optimal input pattern. A summary of the results achieved by Pelikan et al. [103], a fully connected first and second order MOHN and MSDA are shown in figure 4.37. The number of fitness evaluations required to optimise the function are plotted against the number of variables in the function input. In the original paper, Pelikan and Mühlenbein also showed the results for a genetic algorithm but they were so much higher than the figures presented here they have been excluded from the chart. The GA required 140,000 fitness evaluations for the 120 variable version of the function.

It may be possible to reduce the sample size further by iteratively adding to a sample only when the algorithm needs it, rather than using an arbitrarily chosen 50%, but that is a subject for future work. These results illustrate the benefit of being able to swap weights in and out of a MOHN to either minimise the size of the required data set as in this case, or to make the most of a fixed data set without over-fitting.

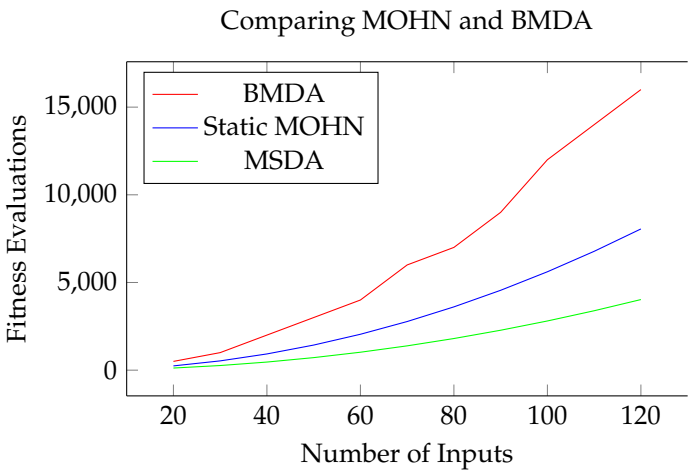

Figure 4.37: Number of fitness function evaluations required to optimise the quadratic fitness function given in [103]. The top line shows the figures for BMDA taken from [103], and the other two are different approaches to training a MOHN.

4.13 Comparing Structure Discovery with Markov Random Fields

This section compares the MOHN used as a fitness function with two Markov random field approaches to building an EDA, DEUM [117] and MARLEDA [2].

First we compare the MOHN structure discovery algorithm with the DEUM clique based structure learning approach described in [117]. DEUM is designed as a search heuristic, which follows the EDA pattern of building a distribution of promising candidate solutions and then sampling from it to produce an improved generation of solutions. Although DEUM uses the EDA framework, it is generally not reported as being used in an evolutionary mode. Rather, it builds a distribution in a single step and then samples that until the solution is found.

The DEUM structure learning algorithm finds all pairwise interactions between variables and uses the resulting graph to infer higher order interactions. A cross entropy measure is used to detect pairwise interactions. Cross entropy between two variables, X_1 and X_2 from a data set $\mathbf{D}$ is

$$CE(X_1, X_2) = \sum_{x_1, x_2 \in \mathbf{D}} p(x_1, x_2) \ln \left(\frac{p(x_1, x_2)}{p(x_1)p(x_2)} \right) \tag{4.16}$$

An edge is connected between any pair of variables with cross entropy above a threshold, TR, which is the average of the cross entropy values of every possible pair multiplied by 1.5. Once the second order connections have been found, the maximal cliques of the resulting graph are found using the Bron-Kerbosch algorithm [20], which recursively builds a set of maximal cliques in a graph. The simple version of Bron-Kerbosch is described in algorithm 19.

Algorithm 19 Bron-Kerbosch Maximal Clique Finding Algorithm

BronKerbosch($\mathbf{C}, \mathbf{X}, \mathbf{E}$)
if $\mathbf{X} = \emptyset$ AND $\mathbf{E} = \emptyset$ **then** report $\mathbf{C}$ as a maximal clique
end if
for $x \in \mathbf{X}$ **do**
  BronKerbosch($\mathbf{C} \cup x, \mathbf{X} \cap N(x), \mathbf{E} \cap N(x)$)
  $\mathbf{X} = \mathbf{X} \setminus x$
  $\mathbf{E} = \mathbf{E} \cup x$
end for

The Bron-Kerbosch algorithm is run by calling the function in algorithm 19 with parameters BronKerbosch($\emptyset, X, \emptyset$) where X is the set of nodes in the graph. The set C builds the maximal cliques, the function $N(x)$ finds all the neighbours of node x and the set $\mathbf{E}$ keeps track of used nodes to exclude them from further consideration.

By definition, every node in a clique is connected to every other, meaning that every node is a singular clique, every connected pair is a clique of two and so on. Maximal cliques are those cliques, C where there are no other nodes in the network that could become a member of C and preserve its status as a clique. The smaller cliques that make up C are the sub cliques of C. At this point, each maximal clique represents a fully connected second order sub graph (like a Hopfield network). The resulting network will have a certain capacity for representing functions, but this capacity could be improved by adding higher order connections. [117] present three options for choosing the higher order connections within a clique, C:

1. Add connections only at the order of the size of the clique, $|C|$, so that each clique has only a single parameter associated with it,
2. Fully connect the nodes in each clique at all orders so that each clique represents a basis over its nodes, meaning each clique contains $2^{|C|}$ parameters,
3. Choose a subset of the $2^{|C|}$ possible connections within the clique

The second option is used in [117] and once the structure is defined, the function is learned using ordinary least squares regression. The paper also imposes a limit on the number of connections any node can have in an attempt to avoid overly large cliques. One problem associated with fully connecting the cliques is that increasing the number of parameters drives up the number of function evaluations needed to learn the values for those parameters. As discussed in section 4.12.2.2, even if the weights turn out to be unnecessary, their presence requires additional data.

4.13.1 *Experiments comparing DEUM with a MOHN*

The maximal clique based approach to structure learning has the advantage of being simple to implement and apply. The only parameters that control the algorithm's behaviour are the thresholds for accepting a connection between a pair of nodes and for limiting the number of connections to a node. It is well suited to functions where second order connections are good predictors of higher order connections. However, it is easy to design functions that it would fail on. Any high order interactions that are not also fully connected at order two would not be found, for example. Functions with large numbers of second order connections but few at higher orders would also cause the algorithm problems, either by creating cliques that are very large, or by failing to find all the second order interactions due to the limit on the number of connections allowed. Imagine the extreme example where a function requires a fully connected second order network (a Hopfield network, in other words). This would lead either to one maximal clique that includes every node in the network or to many of the required second order connections being missed.

It should be clear that the MSDA does not suffer from these same drawbacks. If only second order weights are needed, then it will only include second order weights, without limit on their number. If higher order weights are needed, it can still find them even if they are not predicted by the second order structure. However, MSDA is less efficient than clique finding if the function is well suited to the latter approach due to the overhead involved in sampling and testing weights instead of exhaustively assessing every pair in order.

4.13.2 *Multi-Modal Functions*

To compare the two approaches experimentally requires care as it would be easy to design functions that would suit one better than the other. The first set of experiments, described below, uses randomly generated Hamming similarity based functions, as described in section 4.1.1.3, to compare the two approaches. These functions have a single global maximum and a number of local maxima, all at randomly chosen positions in input space. The rest of the input space maps to an output that is proportional to its distance from the closest maximum. This choice allows many different functions to be tested, all with sparse structure and low order connections, which should suit both approaches equally.

4.13.3 *Experimental Setup*

For each experiment in this set, 100 functions of 20 inputs were built and learned, each with nine local maxima and one global maximum, all randomly placed. Each function generated

2000 training points and 200 validation points, which were used to learn a model structure using the maximum clique approach and MSDA. The resulting structures were then trained on the same sample using OLS (the weights found during the MSDA learning were not used so that the structures both had an identical training regime).

The clique finding algorithm was implemented to generate a parameter for every sub clique in the maximal cliques (option 2 from above). The MSDA used lasso for weight value estimation and removal, and used the hyperparameter settings described in section 4.11.1. Ten initial experiments ran with 600 weights being added before each learning iteration (approximately a third of 2000, as described in section 4.11.1). On noting that the resulting MOHNs contained around 50 weights in total, the full experimental run of the algorithm across 100 different target functions was made where only 50 weights were added at each iteration. The initial runs required 10 iterations of the MSDA so the used weight emptying schedule was set to every 5 iterations on the remaining experiments.

The average root mean squared error on validation data across the 100 functions was calculated for both the MOHN and the clique finding algorithm, as was the average size of the resulting networks and the variance of both of these measures.

4.13.3.1 *Results*

Figure 4.38 shows the mean, standard deviation, maximum and minimum of the validation RMSE and number of weights in the networks generated by the clique finding algorithm and MSDA. The MSDA is more accurate and produces smaller networks on average.

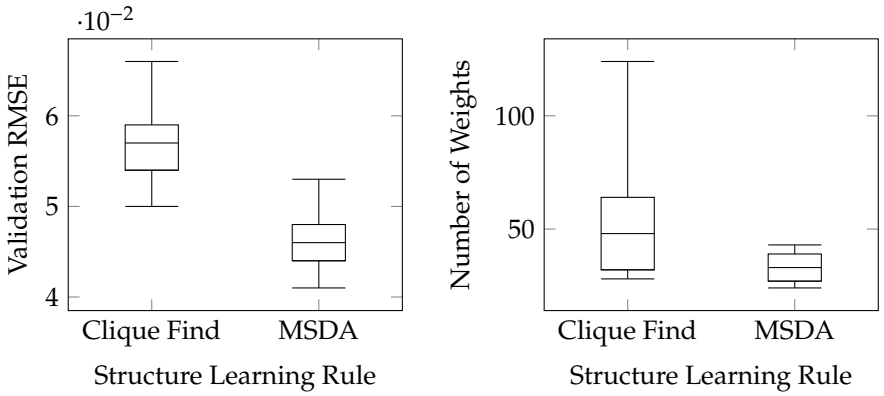

Figure 4.38: Comparing the accuracy and size of models with weights learned using clique finding and MSDA.

4.13.4 *Clique Finding with The Lasso*

One of the limitations of clique based structure learning is that the resulting networks can be too large if the cliques are large. Malago et al. [94] propose a method for limiting the number

of parameters that each clique in DEUM produces by replacing OLS with the lasso as the regression method. The lasso's regularisation forces some parameters to zero, allowing a sparser pattern of connectivity in each clique. In [94], the Ising model was used as a test case, which we will return to later. The Ising model may seem like a poor choice to test this algorithm as all of the cliques are of size two. The only possibility for finding larger cliques occurs if the cross entropy measures used by DEUM to find the pairwise connections choose spurious connections that cause larger cliques to form.

4.13.4.1 *Experimental Setup*

To test whether this approach is more generally applicable, the same experiments using various multiple pyramid functions described above were repeated comparing an approach where the maximal cliques were fully connected at all orders to one where they were only connected where the lasso found non-zero coefficients. 100 trials were repeated with a randomly generated function with nine local and one global maximum. The RMSE and the number of weights in each model was recorded.

4.13.4.2 *Results*

Figure 4.39 shows the mean and variance of the RMSE of the 100 networks built using the fully connected cliques and OLS compared to the RMSE of 100 networks built by training fully connected cliques with the lasso. Figure 4.39 also compares the two methods in terms of resulting network size. The average error in both cases is the same, but the lasso network is smaller on average (40 connections) than the OLS (48 connections).

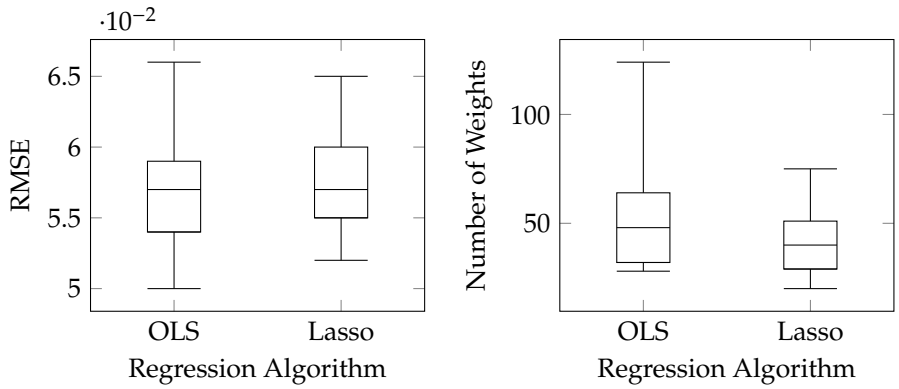

Figure 4.39: Comparing the accuracy and size of models with weights learned using OLS and the lasso from fully connected cliques.

4.14 Ising Spin Glass Learning

The fitness function used to demonstrate the clique finding algorithm in Brownlee et. al [117] was the Ising spin glass problem. Although the Ising model is not an ideal test for structure discovery algorithms as the fitness function has only second order components, [117] provides a set of results with which the MOHN structure discovery algorithm can be compared.

4.14.1 *Learning Structure with The Lasso MOHN*

This section first shows that an Ising model can be learned from a number of fitness evaluations equal to the number of weights in the MOHN. Assume that the fact that the model is second order only is known, but that the connection pattern is not. A MOHN with $n(n-1)/2$ weights is needed, so the data sample should be that size.

4.14.1.1 *Experimental Setup*

In the first set of experiments, a MOHN was used to learn ten randomly generated 10×10 Ising models. With 100 input nodes, the model would have $1 + 100(99)/2 = 4,951$ weights (including ω_0), so the MOHN was trained on 4,951 samples of the Hamiltonian energy function described in equation 4.8. Parameters were learned using the lasso on a network fully connected at order two. Any parameters set to zero by the lasso were then removed from the MOHN. The correlation between the model output and the energy Hamiltonian of the Ising model in the validation data was calculated and the parameters remaining in the model after the lasso pruning were compared to the known structure. The experiment was repeated on ten different randomly generated Ising models.

For comparison, [117] use three samples of diminishing size: the population, $\mathbf{P}$, the selection sample, $\mathbf{D}$ and the parameter learning sample, $\mathbf{L}$ where $\mathbf{L} \in \mathbf{D} \in \mathbf{P}$. $\mathbf{L}$ is used to estimate the parameters rather than $\mathbf{D}$ for the sake of speed, but the number of fitness evaluations used for comparison is $|\mathbf{P}|$.

4.14.1.2 *Results*

With 4,951 fitness evaluations, all ten Ising models were learned by a MOHN with a validation correlation of 1 and all ten contained exactly 200 weights in the correct configuration for the target Ising model. The experiments reported in [117] used a sample of size $|\mathbf{D}| = 5,000$ to estimate the model structure, but used a population of size $|\mathbf{P}| = 30,000$ to select sample $\mathbf{D}$.

4.14.2 *Finding the Optimal Spin Configuration*

Having learned a MOHN that can mimic the Ising Hamiltonian, it should be possible to apply a search method to generate the configuration of inputs that minimises the energy of the learned model. Section 3.5 describes a number of methods for searching a MOHN for an optimal pattern and these were compared for their ability to find the optimal spin configuration of the MOHN Ising model.

4.14.2.1 *Experimental Setup*

A single 100 node Ising model with connections drawn randomly from $\{-1, 1\}$ was used for these experiments. A MOHN was trained using the lasso followed by OLS and an examination of the weights showed them to form a perfect representation of the Ising model in question. The ground state of the network was found using the on line resource from the University of Cologne [1] and this pattern was scored with the Ising model. This score was used as the target for testing the methods for searching the MOHN.

An initial analysis of the search space was carried out by performing RRHC for 5000 restarts and building a histogram of the frequency with which the algorithm visited each attractor. This produced 4989 unique local optima and no global optimum. This shows that RRHC will not be able to solve the Ising model and that LOSS and ILS are unlikely to be applicable in this case due to the number of local attractors. Simulated annealing and weight satisfaction search were chosen for comparison. Each method was run until either the optimal pattern was found or 5000 iterations were complete. An iteration of WSS involves reaching a point where no neuron update can improve the score (a local optimum) and an iteration of SA is one pass through the temperature cooling schedule, also resulting in a local optimum. If no solution was found after 5000 iterations, the algorithm was deemed to have failed.

In the cooling schedule for SA, T started at 200 and was halved every 20 steps of the whole network until it reached 0.003, at which point the network was allowed to settle and the next iteration began. It takes 17 updates of T to get from 200 to 0.003 and 20 steps of the network to make one update, meaning a full pass of the cooling schedule take $17 \times 20 = 340$ network updates. For a network of size n, that equates to $340n$ evaluations if the true fitness function was being used. For a 100 node network, that is 34,000 and for a 400 node network, 136,000 fitness evaluations. If $ns > 1$ iterations of the cooling schedule are required, then these values are multiplied by ns. As simulated annealing could be run on the fitness function itself with no modelling required, these are the targets to beat.

1 http://www.informatik.uni-koeln.de/spinglass/

4.14.2.2 *Results*

Table 4.12 shows the number of iterations required by SA and WSS to find an optimal configuration of spins over 100 trials of the same Ising model. It also shows the number of failed attempts (from 5000 iterations) and the variance of the number of attempts. The average iterations is taken over the number of successful attempts. It is clear that SA out performs WSS significantly. WSS fails 11% of the time and has a large variance in the number of iterations it needs. In this case, simulated annealing is clearly the best choice. It also suggests that, for Ising models at least, it is more efficient to learn and search a model of the function than to build an EDA that needs a larger training sample. The simulated annealing took an average of 2.5 iterations of the cooling schedule, so the number of fitness function evaluations it would have needed is $2.5 \times 34,000 = 85,000$. The MOHN modelling process has reduced the number of evaluations considerably, using only 4989 evaluations.

Method	Iterations	Variance	Fails
SA	2.5	1.8	0
WSS	1,666	1,262	11

Table 4.12: Average iterations of simulated annealing and high order search on a MOHN representation of an Ising model.

4.14.3 *Discovering Ising Structure*

A number of aspects of the previous experiment can be improved upon. The second order weights were all added in a single pass, and no higher order weights were considered (a piece of prior knowledge that might not be available). A set of experiments were carried out to address these issues where the weights were allowed to take any order from 1 to 5 and new weights were added in smaller increments.

4.14.3.1 *Experimental Setup*

The same 10×10 Ising structure as described above was used to generate a sample of data to train a MOHN containing 3000 samples generated uniformly at random and evaluated by the Ising Hamiltonian. The MSDA was run with an initial 1000 weights and with 1000 added at each iteration. The learning method was SGD and the rest of the MSDA hyperparameter values were set as described in section 4.11.1. The stopping criterion was a RMSE of less than one (which corresponded to a correlation between the model output and the target function output of over 0.99). An initial run of the algorithm suggested that the used weight list should be emptied every 15 iterations.

4.14.3.2 *Results*

Figure 4.40 shows the training and validation RMSE of the MOHN by epoch as it learned. The error trace reflects a number of properties of the algorithm. The reduction in error during gradient descent is visible between peaks that show the points where weights are discarded and new weights are added. It is also clear from the trace that although these changes in weights cause the error to spike, they do not take the error back up to the point it was at when the network was new. This is evidence of the efficiency of the gradient descent approach over learning each network from scratch at each iteration. Validation error is greater than training error while the MSDA is searching for the correct weights, which represents model bias, but as more of the required weights are found and model bias diminishes, the gap between the training and validation error closes until, at the point where the model is correct, they converge. There is no model bias (as we will verify when we look at the weights, next) and minimal estimation bias, which can be removed entirely using OLS if required.

Figure 4.41 shows a trace, by iteration, of the number of weights at each order from 1 to 5 in the network as it learned, with the top line showing the total number of weights. Note the spikes at iterations 15 and 30 where the history of weights to avoid was emptied and the way the number of second order weights grows as the rest reduce in number in the second half of the training. At the final step, the number of second order weights reaches 200 (which is the correct number) and the rest all drop to zero, which is also correct.

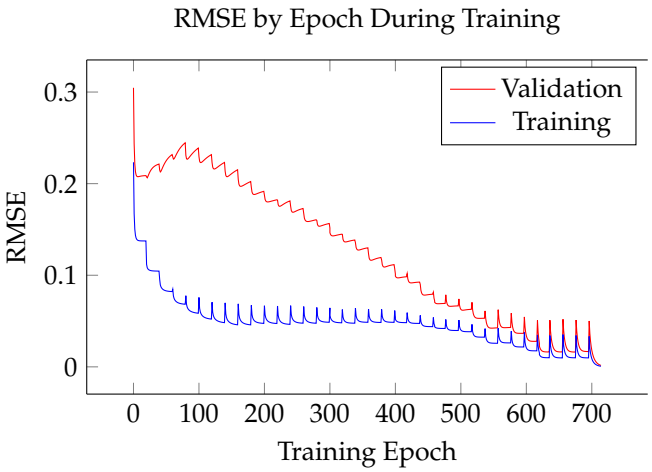

Figure 4.40: Training and Validation RMSE during MSDA learning of a 100 node Ising model from 3000 training samples.

The algorithm has found the correct structure, and achieved a correlation with the target function output of 1 using only 3000 evaluations of the fitness function. This figure is only a tenth of the 30,000 required by [117] and is even fewer than the number of possible second order interactions in the network. It was done without the prior knowledge that weights need only be at order 2. A clique finding method that relies on comparing every pair could not work

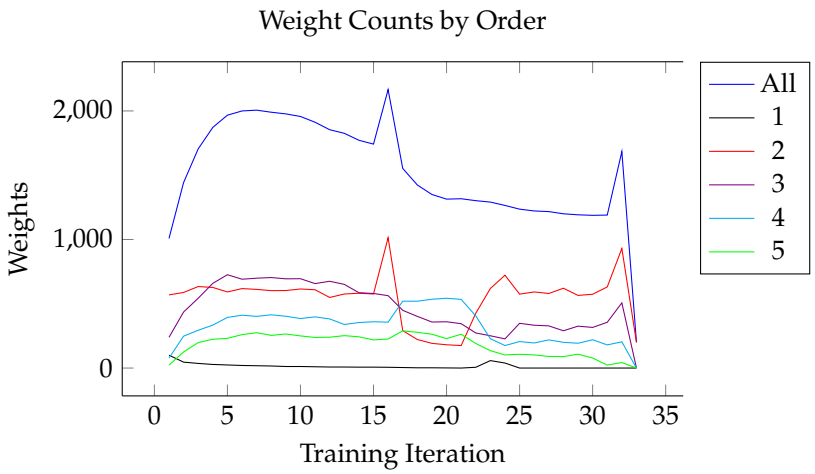

Figure 4.41: Weight counts at different orders during MSDA learning a 100 node Ising model.

with so few samples as there are 4950 possible pairs to consider. The MSDA is able to use fewer evaluations as it considers subsets of weights in the context of a partial model. This is why the history of weights to avoid needs to be deleted occasionally as the context in which those weights were first discarded is different from the context later in the training process.

4.14.4 *Reducing the Sample Size Further*

The required sample size may be reduced further by not requiring the network to discover the relevant weight orders. If the knowledge that the weights are second order only is used (or the second order interactions are all that are required as step one of a clique finding approach) the same process can be run with smaller samples.

4.14.4.1 *Experimental Setup*

The same Ising model was used again in this experiment. The sample of fitness evaluations was limited to 2000 and the order of weights in the MOHN was restricted to two. As before, the MOHN was initialised with 1000 weights and had up to 1000 added at each iteration, ensuring that the total number of weights remained below the training set size of 2000. The used weight list was emptied every 20 iterations.

4.14.4.2 *Results*

The training process was longer than that for previous experiments, needing a total of 90 iterations of the algorithm. The correlation between model and target did reach 1, however. Figure 4.42 shows the error trace by epoch and figure 4.43 shows the weight count by iteration (the weight count does not change by SGD epoch, but the error does). An interesting aspect of figure 4.43 is that the weight count hovers around 650 for many iterations before suddenly

dropping to 200, the correct amount. The error is diminishing during this time, but it is only when the last required weight is included that the others are able to lose significance and be dropped.

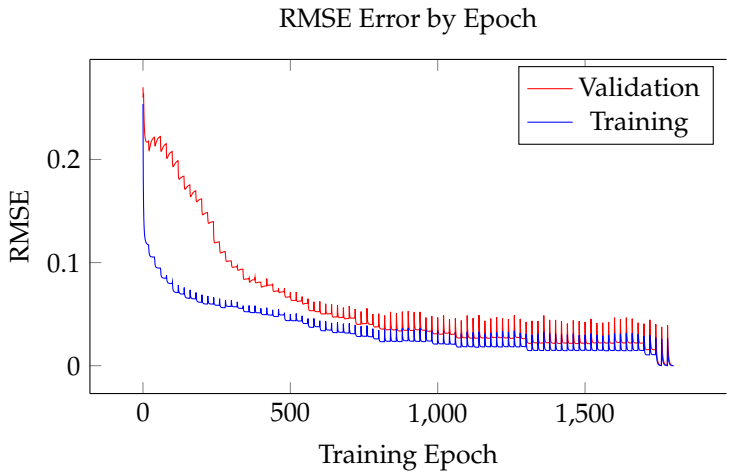

Figure 4.42: Training and Validation error during MSDA learning of a 100 node Ising model with 2000 data points. The MSDA was limited to searching first and second order weights only.

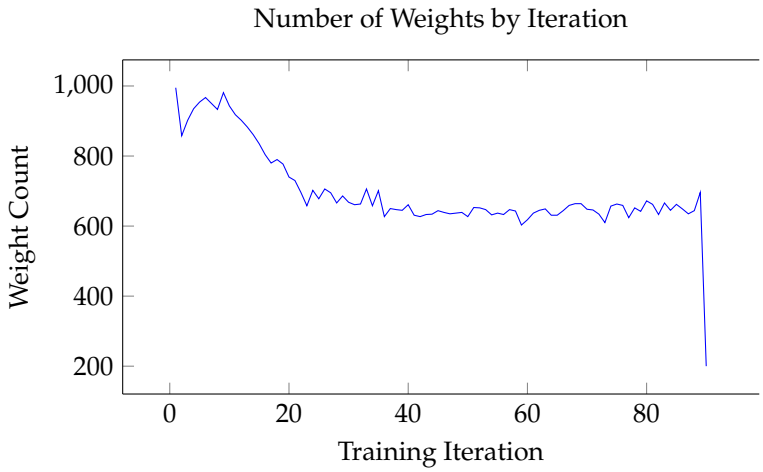

Figure 4.43: Weight count during MSDA learning a 100 node Ising model at order two with 2000 data points.

Figure 4.44 shows the 100 node Ising model learned by a MOHN presented in the visualisation method described in section 3.6. Each column is a node and each row represents a weight. Each row has two pixels, representing a second order connection. The nodes in the MOHN vector, X represent the square structure of the Ising lattice so that $X_1 \dots X_{10}$ are the first row of the lattice, $X_{11} \dots X_{20}$ are the second, and so on. Figure 4.44 clearly shows the connections between adjacent pairs that are the right hand neighbours, the connections 10 bits apart that represent the neighbours below and the few distant connections that represent the wrapped connections of the torus.

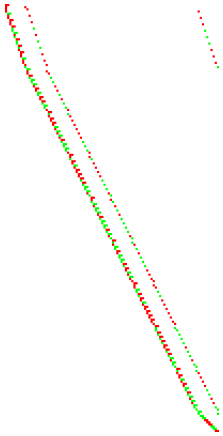

Figure 4.44: The weights of a 100 node Ising model learned by a MOHN using structure discovery. Each row represents a weight and its connections to nodes, which are represented in columns. The resulting image has three diagonal lines of connections. The left hand line shows nodes connected to their immediate horizontal neighbour. The middle row shows the vertical connections to a node below and the final, smaller line shows that the top row of nodes is connected to the bottom row.

Figure 4.45 shows in more detail the relationship between the weight chart and the Ising model. With a little practice at reading them, the weight charts can become fast and convenient tools for checking the structure of a MOHN once it is built. In regular structures such as the Ising model, it is also quite easy to spot any missing or spurious weights by studying the chart.

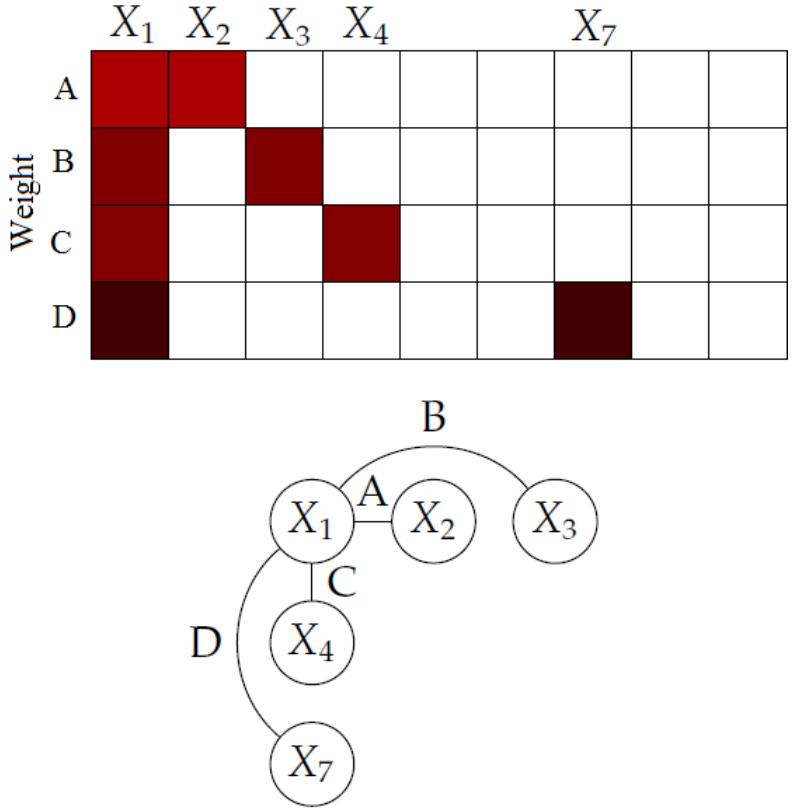

Figure 4.45: Detail of the weight chart and connections from a single node, X_1 in a nine node 2D Ising model. Compare this to the top rows of figure 4.44 to see where the connection from top to bottom (D in this figure) is shown.

4.14.5 *A Larger Network*

Brownlee et al. [117] also present results for Ising models of 256 and 400 nodes. The final experiment in this section describes the MOHN structure discovery algorithm working on a 20×20 Ising model with 400 nodes. The algorithm was given a sample of 20,000 evaluations of the 400 node Ising Hamiltonian and run with 6,000 weights being added at each iteration. Parameter estimation was by SGD and all the MSDA hyperparameter values were set as described in section 4.11.1 and the stopping criterion was a correlation with the target output in the validation set above 0.99. The algorithm required 195 iterations to find the correct structure. Table 4.13 summarises the results of the Ising model experiments comparing DEUM to a MOHN.

Method	Size	Evaluations
DEUM	100	30,000
MOHN-Lasso	100	5,500
MSDA-Order 5	100	3,000
MSDA-Order 2	100	2,000
DEUM	400	150,000
MSDA	400	20,000

Table 4.13: Comparing the number of fitness function evaluations used to learn Ising models of 100 and 400 nodes using DEUM and MOHN structure discovery.

4.14.6 *Comparing MOHNs to MARLEDA*

Another Markov random field approach, MARLEDA [2] has used the Ising model to demonstrate its capabilities. MARLEDA uses a very similar approach to DEUM in that it uses a MRF. Pairwise interactions are found using Chi squared rather than cross entropy. Alden and Miikkulainen [2] report some recent results on similar experiments comparing its approach to a standard GA using GENEsYs [6] and the Bayesian optimisation algorithm (BOA) [102]. They compare results from a version of MARLEDA that learns the function structure with a version that is given the structure and only needs to learn the parameters. Results are presented for a 400 node Ising model, as here, but the process is stopped after 20,000 fitness evaluations, at which point only the MARLEDA with the given model structure is able to reliably find the global optimum. Alden and Miikkulainen [2] report that the version of MARLEDA that also performed structure learning found only solutions that scored 80%-85% of the global optimum. They report "The deceptive qualities of this domain were not completely overcome". Based on the analysis presented in the experiments in this section, it seems likely that MARLEDA would have needed more fitness evaluations to find the correct model. For a 400 node Ising model, the minimum sample required to model all of the second order interactions is $(400 * 399)/2 = 79,800$. The MSDA was able to discover the correct structure (unlike MARLEDA, which needs the structure to be defined) with 20,000 fitness evaluations. The GENEsYs GA and BOA are both reported in [2] to perform worse than MARLEDA, and so worse than the MOHN.

4.14.7 *Comparing MOHNs to sDEUM*

An alternative approach to choosing which weights to include in a DEUM model was proposed by Valentini et al. [139], who used the lasso to set unused weights to zero in an approach they

called Sparsified DEUM (sDEUM). They presented results comparing standard DEUM, sDEUM, simulated annealing and hBOA given the task of finding the global optimum in a 3D Ising model. A 3D model extends the neighbourhood of each node to those other nodes that would be its neighbours in a cube. The largest model analysed in [139] contained $25 \times 25 \times 25 = 125$ nodes and it is that size of network that is used to compare the performance of a MOHN. sDEUM required an average of around 20,000 evaluations of the 125 node Ising Hamiltonian to find an optimal input pattern. The next experiment attempts to solve the same problem in 5,000 evaluations.

4.14.7.1 *Experimental Setup*

The MSDA was used to discover the correct structure of randomly generated 3D Ising models of 125 nodes. 20 repeated trials of the experiment were carried out to verify that the results were robust. For each trial, a training set of 5000 examples was generated, consisting of uniformly random input patterns and their associated output from the Ising Hamiltonian. The MSDA regime from previous Ising experiments was kept, which meant adding 1000 weights at each iteration and using the default hyperparameter settings listed in section 4.11.1. The learning process was stopped when the correlation between the MOHN output and the validation data from the target Ising model was greater than 0.99. A total of 20 trials with random networks were repeated and the results averaged, but an initial run was made to provide a clue as to where the used weight list should be emptied. This run suggested that 50 iterations would be a good interval as it began a plateau of training and validation error.

4.14.7.2 *Results*

A MOHN trained on 5000 samples from the Ising energy function was able to find the correct structure and weights in an average of 51 iterations of the MSDA. This suggests that the emptying of the used weights list (done at 50 iterations) was important in allowing the algorithm to find the correct weights. The resulting correlation between Ising model and MOHN on validation data was always 1. Figure 4.46 shows an example trace of the root mean squared training and validation errors during training. Figure 4.47 shows the weight counts at each order. Note that the total number of weights is always below half the training sample size, meaning the model is not overfit during learning. Figure 4.48 reproduces part of figure 3 from [139] showing the results reported in that paper for a 125 node Ising model with an additional entry for the MOHN.

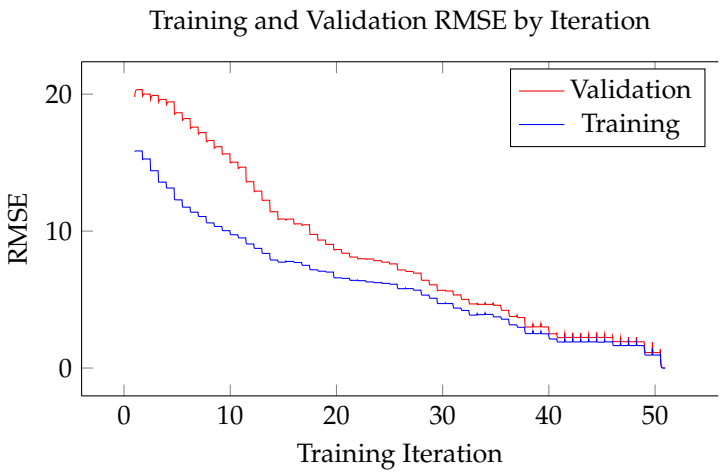

Figure 4.46: Training and validation error during MSDA learning of a 125 node 3D Ising model with 5000 data points.

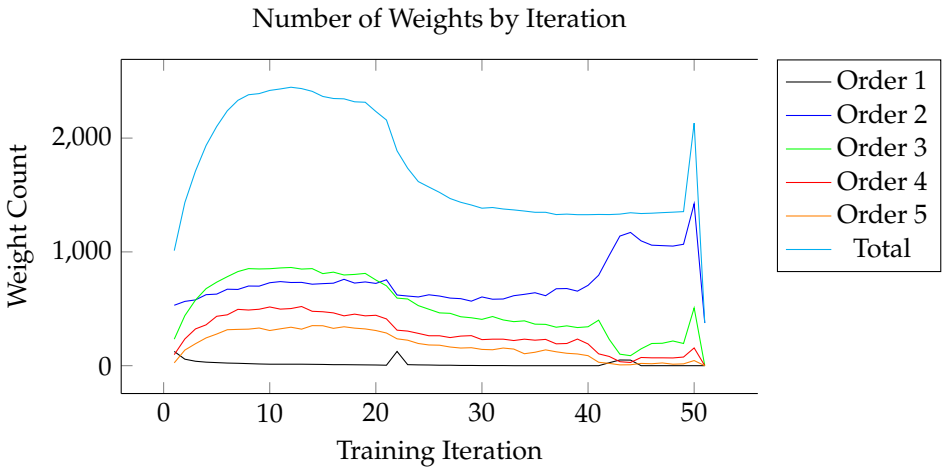

Figure 4.47: Weight count during MSDA learning of a 125 node 3D Ising model with 5000 data points.

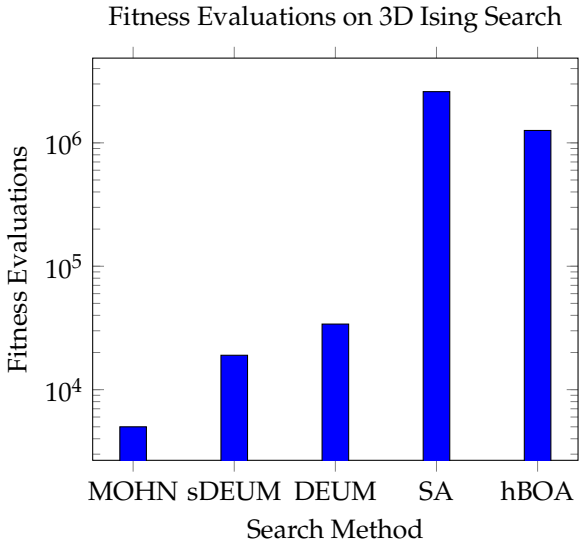

Figure 4.48: Average number of fitness evaluations (log scale) required to find the first optimal solution to a 3D Ising model by different algorithms.

4.14.8 *Learning Ising Models with an MLP*

A MOHN was able to find the correct structure for 2D and 3D Ising models with small sample sizes. This section addresses the question of whether an MLP would be a suitable alternative. If an MLP could be trained in fewer samples, it could be searched using simulated annealing and provide a solution in fewer fitness evaluations.

A 100 node Ising model was used for these experiments. The MLP that was used had a single linear output unit, and a grid search was employed to find suitable hyperparameters values. The search was across all combinations of the following hyperparameter value sets:

Hyperparameter	Grid Set
η	{0.1,0.2,0.4}
η decay	{1,0.9,0.8,0.5}
hidden units	{80,120,180,300,600,1000}
hidden layers	{1,2,3}
Initial weight range	{0.01,0.1,1}
momentum	{0.1,0.5,0.8}
Hidden activation	{ReLU, Tanh, Logistic}
Mini batch size	{1,10,50,100}

The first experiment used a data set containing 20,000 training examples. This figure is higher than the target we are aiming for, but the first set of experiments are designed to establish an effective architecture and training regime without the uncertainty associated with having a training set that is too small.

The error trace of the first four combinations in the grid search over 500 training epochs showed the validation RMSE flattening after 150 epochs. This figure was used as the stopping criterion for the training regime during the grid search, with the intention of exploring longer training regimes once the hyperparameters were fixed.

The results of each combination in the grid search were sorted by validation error and the top ten (the ten with the lowest validation error) were selected to guide a further refinement of the search. All of the top ten networks had 1 layer of tanh activation units, an η value of 0.1 or 0.2, no learning rate decay and a batch size of 1 (i.e. SGD). The momentum and starting weights range values were uncorrelated with the error rate. Networks with 300, 600 and 1000 hidden units all featured in the top 10. The validation error for the top ten networks ranged from 0.055 to 0.133, which are very high but the short training time may explain that.

A narrower grid search was performed over networks with 300, 600 and 1000 hidden units, all with tanh activation functions and learning rates in {0.1,0.2}, momentum of 0.5, no learning

rate decay, a batch size of 1 and random starting weights in a range of 0.1. In this search the number of training epochs was increased to 200,000. The best network achieved a validation score of 0.044 with 1000 hidden units. Another attempt was made with an MLP with 2000 hidden units, keeping all other hyperparameters at the same level, and it achieved a validation RMSE of 0.034, but with considerably longer training time.

At this point, having spent a lot more time exploring MLP hyperparameters than was spent training many MOHNs to a validation error of zero, the search was terminated. Reducing the validation RMSE further is almost certainly possible, but these experiments were with 20,000 examples and the aim was to build an MLP based on fewer than 3000, which is what the MOHN required. For those reasons, we conclude that further experiments with the MLP are not helpful and that in this case, the MOHN is preferable. It is true, of course, that the MOHN matches the Ising model very well in terms of structure and we would expect a MOHN with the right structure to reproduce an Ising model perfectly. Some fitness functions suit a MOHN well, others will suit an MLP very well. Our claim is that a MOHN provides a method worth including in a fitness function modeller's tool kit for the times when it can be discovered that the function is well suited to its abilities. The Ising model provides a benchmark example of such a function.

4.15 Comparing MOHNs to Boltzmann Machine EDAs

Boltzmann machines are a type of neural network that use a stochastic activation function that enables them to model probability distributions. They represent dynamic systems that can be used to generate data in a Boltzmann distribution using Gibbs sampling. Both deep Boltzmann machines [108] and restricted Boltzmann machines [109] have been used to build EDAs for combinatorial optimisation. This section compares the results from Probst and Rothlauf [108] and Probst et al. [109] with results from using a MOHN to model and search a single fitness function: the k-bit trap.

Both Probst and Rothlauf [108] and Probst et al. [109] present results for searching k-bit trap problems of various sizes with Boltzmann EDAs, reporting both the number of fitness evaluations required to find a solution and the CPU time taken. In [108], deep Boltzmann machines were used in a method called DBM-EDA and in [109], the RBM-EDA used restricted Boltzmann machines. Both papers compared the performance of the EDAs to that of a Bayesian optimisation algorithm (BOA) on a number of problems including the k-bit trap.

Unlike the results reported above for DEUM, the Boltzmann EDAs make use of a number of generations of a cycle of data generation, selection and modelling to perform an evolutionary search. The principle behind an evolutionary EDA is that the model can be simpler as only the space of promising (and eventually, very good) solutions is modelled. The risks associated with

the evolutionary approach are that larger samples from the fitness function may be needed to build multiple populations. This is in contrast to the approach of building and then sampling an accurate model reported in [94] and [117] and employed by the MOHN model and search approach.

4.15.0.1 *Experimental Setup*

Some of the RBM-EDA and DBM-EDA experiments were repeated using a MOHN as a fitness function model. Specifically, the k-bit trap problems for 4 and 5 bit traps were modelled and searched using a MOHN. The number of fitness evaluations and the time taken to model and search each problem was recorded. The MSDA was used with the same settings for every trial. The SGD learning rule was used with the settings given in section 4.11.1 and the used weights list was emptied every 20 iterations of the algorithm. The number of samples to use for learning was fixed for each experiment based on the size of the problem and the number of samples reported in [108]. Each fitness function was modelled 10 times and the average time and number of fitness evaluations was recorded.

4.15.0.2 *Results*

In all cases, the MOHN was able to model and successfully search the fitness function in far fewer evaluations and in much less time than the results reported for RBM-EDA, DBM-EDA and BOA. Table 4.14 summarises the results, taking data from [108] and [109]. Note that the results from the DBM-EDA are for trials where the global optimum was found 90% of the time or more. All other results provide numbers where all searches found the global optimum. The BOA and DBM-EDA figures were taken from table 1 in [108]. The figures for RBM-EDA are approximate as they were read from the graphs in figure 3 in [109].

Problem	Algorithm	Evaluations	Time
4-trap	BOA	13,673	2,728
40 bits	DBM-EDA	47,231	2,201
	RBM-EDA	16,000	150
	MOHN	2,000	22
4-trap	BOA	43,777	43,935
80 bits	DBM-EDA	153,278	13,271
	RBM-EDA	160,000	1,100
	MOHN	10,000	170
5-trap	BOA	14,924	1,384
25 bits	DBM-EDA	13,291	566
	MOHN	1,000	8
5-trap	BOA	47,904	20,199
50 bits	DBM-EDA	49,886	3060
	RBM-EDA	63,000	300
	MOHN	20,000	295

Table 4.14: Unique fitness function evaluations and time required to find the global optimum in different k-bit trap functions using a MOHN and the figures presented in [108] and [109]. No data is available for the RBM-EDA performance on the 5-bit trap problem over 25 bits.

The models built by the MOHN were searched using weight satisfaction search (see section 3.5.2), which was able to find the global optimum in a single pass of the algorithm. This is very fast — in every case it took less than an additional second to search the model once it had been successfully built. Of course, this function is perfectly suited to the WSS as each trap is small and can be solved independently.

4.15.1 *Learning the K-Bit Trap with an MLP*

This section compares the use of MLPs to learn and search the same set of k-bit trap fitness functions described above. The questions of interest concern the smallest number of samples an MLP would require to accurately reproduce a k-bit trap function, the variance of results across an ensemble of such solutions, and methods for using the MLP to search for an optimal input pattern.

4.15.1.1 *Experimental Setup*

The search for an MLP capable of learning a k-bit trap function with the smallest number of training data points was carried out by first attempting to optimise the hyperparameters of the MLP structure and training regime on a training set that was assumed to be large enough, guided by the training set sizes reported in table 4.14. Using 60,000 uniformly random input patterns and the result of evaluating each with a 4-bit trap function over 40 bits (those used for the first set of results in table 4.14), a grid search was performed over the following hyperparameter space:

Hyperparameter	Grid Set
η	{0.1,0.2,0.4}
α	{0.8,0.5}
Hidden activation	{Tanh, Logistic, ReLU}
Number of hidden units	{10,20,30,40}
Number of hidden layers	{1,2,3}
Random weight range	{0.01,0.1,1}
Mini Batch size	{1,5,10,20,40,80}

All MLPs used a single linear output unit and the output data were scaled to fall within the range from zero to one. The grid search was ordered so that smaller numbers of hidden units were tried first, and stopped when more than three networks of the current size achieved a correlation over 0.99 as there is no requirement for a larger network if a smaller one can find the solution. Ten hidden units were found to be insufficient, but networks with 20 hidden units were able to achieve a validation correlation of 1.

As a smaller network should require fewer training examples, a second search was performed to attempt to reduce the hidden unit count further, using a single hidden layer of sizes from 11 to 19 and the rest of the hyperparameters set as follows, based on the results of the first grid search:

Hyperparameter	Grid Set
η Learning rate	0.1,0.2,0.4
Learning rate decay	None
α Momentum	0.5
Hidden activation	Logistic
Hidden Layers	1
Hidden units	{11,12,13,14,15,16,17,18,19}
Random weight range	0.26
Training epochs	500
Training samples	60,000
Mini Batch size	1 (Simple SGD)

The only two parameters that were searched were the learning rate and the number of hidden units. The goal was to find the largest learning rate that was stable for the smallest number of hidden units. This was found at networks with 14 hidden units and a learning rate of 0.2, which are the values that were used for the rest of the experiments described in this section. The variance of validation correlation for the chosen hyperparameter set was tested by generating 20 random data sets of size 60,000 and training an MLP on each. The test correlation between the MLP output and the known function output varied between 0.73 and 0.99 across the networks built.

The selected hyperparameter set was used for the attempt to minimise the training set size, which was performed with a simple search, starting with a training set size of 1000 examples (half that required by the MOHN) and increasing by doubling the size up to 36,000. The test set consisted of 2000 randomly generated examples. Twenty random training sets were generated for each size in an attempt to account for variation from one MLP to the next. The results showed that a sample of 16,000 data points was the first to produce a test correlation above 0.99. To narrow the search further, the process was repeated with training samples starting at 9,000 and increasing in increments of 1,000 until a training set size was found that consistently gave a test correlation over 0.9. The smallest training set to achieve this contained 14,000 training points. The MOHN value of 2,000 training points required compares very well to this.

Note that SGD, with a batch size of 1 was chosen over a mini batch approach. This was because more SGD trained models found correlations over 0.99 during the grid search than any of the mini batch approaches. However, the optimal mini batch, of size 20, had a lower variance of test correlation than the SGD solutions so the option to train using that size of mini batch was investigated further. Two sets of twenty additional MLPs were trained using the settings described above, one using SGD and the other a mini batch of 20.

Batch size	CC Mean	CC Variance	CC Max	CC Min
Mini batch	0.905	0.0005	0.94	0.86
SGD (batch size = 1)	0.907	0.006	1	0.73

Table 4.15: Comparing the mean, variance and maximum of the validation correlation when training MLPs on the k-bit trap problem with mini batches of 20 compared to training with SGD (batch size of 1).

Table 4.15 compares the mean, variance and maximum of the validation correlation of the MLPs trained with mini batches of 20 and those trained with simple SGD (batch size of 1). There is no difference between the average correlations between mini batch and SGD (p=0.9 from a t-test) but there is a difference in variance, which also results in a difference in maximum correlation. SGD was chosen for the continued experiments with this data because of the observed ability to reach a validation correlation of 1, even though the variance across models was higher.

Each of the 20 MLPs for each batch size was trained on the same training data, so the only difference from one example to the next was the random starting position of the weights. Both batch sizes settled in local error minima (or, perhaps on large plateaux) but SGD showed better ability to escape them at a cost of greater variance across trials. The MOHN, by comparison trained using MSDA was able to produce a correlation between predicted output and actual output on validation data of 1 for every trial. In conclusion, this experiment found that the MOHN was able to learn the 4-bit trap function over 40 inputs in fewer training samples, with lower validation error and lower variance across models than the MLP.

4.15.1.2 *Searching the MLP*

MLPs have been used as surrogate fitness functions, but the motivation in many cases has been to replace a costly fitness function with a model that is faster to evaluate [135]. However, the partial derivatives of the output with respect to each of the inputs to an MLP are easy to compute, meaning that gradient based methods may be employed to search the MLP representation of the fitness function. This is a simple version of the approach used in deep networks to extract images that maximise a class score [119], or to invert the network function [93].

The partial derivative $\frac{\partial Y}{\partial X_i}$ represents the local gradient of the network output in the direction of input X_i at the current input point. It is calculated using the chain rule to calculate the partial derivative of each non-input neuron in the network as the weighted sum of the derivatives from below.

Let a_j represent the activation of neuron j in the first hidden layer and $u_j = f(a_j)$ be its output, then the partial derivative of u_j with respect to input x_i, which are connected with a weight value of w_{ij} is

$$\frac{\partial u_j}{\partial x_i} = w_{ij} f'(a_j) \tag{4.17}$$

Neurons in following layers, including the output are calculated as the weighted sum of derivatives from below multiplied by the derivative of their activation function at their current activation. Now let u_k be the output from a neuron above the first hidden layer:

$$\frac{\partial u_k}{\partial x_i} = \sum_j w_{jk} f'(a_k) \tag{4.18}$$

where the sum is over all neurons in the layer below that containing neuron k.

From a random starting point, a gradient based search may be employed to search for an input that generates a desired value at the output. Random restarts may be required if there are local optima in the function.

The MOHN surrogate models of the k-bit trap problem were searched very efficiency using weight satisfaction search, which is able to take advantage of the fact that only combinations of nodes with weights among them need to be searched. This allows a MOHN representing a function with the structure of a k-bit trap (for example) to be searched very quickly by making changes to more than one input at a time. This final experiment uses a gradient based search to attempt to optimise the inputs to an MLP model of a k-bit trap problem.

4.15.1.3 *Experimental Setup and Results*

The best of the MLP models of the 4-bit trap over 40 inputs was used to guide a hill climbing algorithm in an attempt to find a global maximum output. The hill climber chose a uniformly random starting point and climbed by repeatedly calculating the partial derivatives of the output with respect to each input and changing a single input value in the direction indicated by the gradient. As inputs are binary, that means ensuring the value of the chosen input matches that of the derivative. Inputs were picked uniformly at random with replacement and the derivatives were re-calculated after every change of input value. When no further changes are possible, the algorithm terminates. This process is repeated from different random starting points. As expected, the deceptive nature of the trap function guides the search towards local maxima in which all of the values in a block have a value of zero. Only blocks in which the number of 1s is 3 or 4 in the random start pattern are able to climb to or remain in a global maximum state where all 4 values are set to 1.

4.16 Conclusions

This section has shown that a MOHN can learn a zero error representation of a fitness function and find optimal patterns from it in fewer function evaluations than those reported in some of the literature for EDAs. In the case of the quadratic example from [103] and the k-bit trap problem, a weight satisfaction search was found to be sufficient to find the global optimum from the MOHN model. For Ising models, this approach was less reliable and simulated annealing was found to be effective.

The combination of fast learning from small data sets and guided model search makes MOHNs an attractive option for search and optimisation. Building a full model will never require more data than building a full EDA model and has been shown here to require considerably less in a number of cases. A full model also has the advantage over a partial, evolved EDA that it can be re-used in situations where a number of different solutions need to be generated or a nearest local solution needs to be found quickly.

No single MOHN search algorithm has been found to work well across all the functions tested. In some cases different algorithms were tried and the most successful one chosen, for example when searching the Ising models. In other cases, looking at the weights of the MOHN suggested a sensible choice of search algorithm, such as weight satisfaction search for k-bit trap problems where blocks of separate input sets of clearly defined.

4.16.1 *Further Development*

The MOHN experiments were not as dynamic as they could be in the sense that the sample sizes were fixed. An approach that adds both weights and samples as structure discovery proceeds might yield even better results. Such an approach would start to become population based in the sense that new data (i.e. a new population) would be added at each iteration, allowing the model to grow more complex as the sample size grew.

Similarly, the method described here took a two stage approach to search, which involved building an accurate model and then searching it. An integrated approach where a model is searched as it grows might find a maximum sooner if a poorer model leads to it, or it might take longer if the overhead of searching the model outweighs the gain from stopping early. Further work is required to address these questions.

Part III

SUMMARY AND CONCLUSIONS

5

CONCLUSIONS AND FUTURE DIRECTIONS

MOHNs have been presented as regression models with a structure that allows human interpretation, guided heuristic search and informed regularisation. They can also be used as content addressable memories with arbitrary capacity. This final section considers future directions for MOHN research and presents a conclusion.

5.1 Future Directions

This thesis has concentrated on functions of the form $f : \{-1, 1\}^n \rightarrow \mathbb{R}$. That means a restriction to binary valued inputs and a single, real valued output. The following sections suggest further research directions for work on MOHNs that relax these constraints and show some speculative and very preliminary results in those directions.

5.1.1 *Real Valued MOHN*

This work has restricted its attention to MOHNs with binary valued inputs. The main reason for this is the fact that a full MOHN forms a basis for functions in $f : \{-1, 1\}^n \rightarrow \mathbb{R}$. It also allows comparison with Hopfield networks as content addressable memories and simplifies the search algorithms. The method would be more broadly applicable if it were extended to cover functions in $f : \mathbb{R}^n \rightarrow \mathbb{R}$.

The MOHN learning methods (OLS, the lasso and SGD) all work on real valued inputs so at the simplest level, the MOHN architecture presented here could be used to learn a function of $\mathbb{R}^n$ if the MOHN structure was given. For structure discovery, the lasso could be used for each iteration of the MSD algorithm as the unnecessary weights are automatically set to zero, allowing the algorithm to learn, prune and add as it does in its present form.

The greater challenge in learning a function of $\mathbb{R}^n$ is the fact that a sum of products among subsets of inputs does not form a basis. Take for example $f(X) = sin(X)$ where X is a random variable. A univariate MOHN has only two weights and could not capture such a relationship. A solution would require either hidden units such as those in an MLP or an expansion, for example a power series $f(X) = \omega_0 + \omega_1 X + \omega_2 X^2 + \ldots$. A full solution would require the ability to add nodes to the network for new variables representing a series (X^2, X^3 etc.) in

addition to the current structure discovery approach in which higher order interactions might involve X^2X^3 or $X_1^2X_2^2$ for example.

5.1.1.1 *Experimental Examples*

Two small experimental examples are given here to illustrate the ideas. In the first experiment, the univariate function $f(X) = \sin(3X) + \sin(6X)/2$ is learned in the range $-1 \leq X \leq 1$ by allowing the MOHN to have 8 nodes: one to represent the value of X and the others that are given the values $X^i, i = 2 \ldots 7$. The MOHN was given first order weights only and trained on a random sample of 1000 instances of $X, f(X)$ using the lasso and achieved a correlation coefficient between the target and the predictions of 0.993. The weights revealed the function learned by the network to be approximately $5.7X - 19X^3 + 21X^5$, the powers not included having been forced to zero by the lasso.

In a second experiment, the function $f(X) = 1.3X_0X_1 + X_2 - 3X_2X_3$ was learned with a first and second order MOHN of 4 nodes, again using the lasso. In this example, no inputs were allocated to X^2 etc, but all interactions between pairs of variables were considered. The resulting MOHN had 11 possible weights and set them all to zero except those corresponding to those in the target function.

5.1.2 *Heuristic Optimisation*

The experimental results comparing a MOHN to other EDA approaches to optimisation are encouraging and require further investigation. The learning and searching phases are currently separate, but it would be interesting to attempt to combine them into a single process with the goal of finding an optimal solution with the fewest fitness function evaluations. For example, it may be that some functions yield an optimal solution before the validation error reaches its minimum. The experiments on content addressable memory and model bias reported in section 4.5.5 suggest that this can be the case for some functions.

Two questions raised by this work are *How can the structure of the MOHN guide a heuristic search?* and *Which functions are amenable to a MOHN optimisation approach?* Section 3.5.0.2 proposes some ways in which local search can be guided by the weight structure of the MOHN, for example making the perturbations of iterated local search based on the weight values, but the effectiveness of the approach depends greatly on the structure of the network. It is very effective for the MOHN that represents a k-bit trap, but does not add much to the search of a MOHN model of an Ising network. Other methods from the literature such as those proposed by Whitley et. al. [143], [29], [137], propose fast methods for searching variable interaction graphs such as a MOHN. Their effectiveness for a range of MOHN functions needs to be investigated. There are other optimisation problem frameworks that use a set of weighted constraints to define a function, for example weighted Max-Sat problems [56] are defined by

a weighted set of disjunctions over subsets of X. Further work is needed to apply the most efficient weighted Max-Sat solvers to MOHN optimisation.

Some functions can be optimised in fewer evaluations than are required to train a MOHN. For example, consider a function that counts the number of inputs with a value of 1 and outputs the Hamming distance between the input vector and either a vector where every input has a value of 1, or one in which every output has a value of -1, which ever is smaller. The function is minimised where all the inputs have the same value. A simple hill climbing algorithm is guaranteed to move to one or other of the solutions, which ever it is closest to at the start point. To learn the function fully, however requires a fully connected MOHN.

Other functions can be learned in fewer evaluations than current state of the art search methods require to optimise them without a model. This is the premise on which the practice of building fitness function models is based. Hybrid approaches that use a model free search on one hand but use the points visited to contribute to building a MOHN should be investigated. Where fitness function evaluations are expensive, the overhead of building and searching a model may be small compared to continuing a search that make more evaluations than the MOHN would need. As fitness function evaluations are often noise free, MOHNs for modelling such functions can be restricted in size to match the number of available data points (evaluations so far).

5.1.3 *Other Possibilities*

The structure discovery learning algorithm could be made more sophisticated. For example, there may be local patterns among subsets of inputs that are repeated and a suitable pattern matching approach may be able to find them. The simpler approach of designing a good user interface to the learning process may allow a mixture of MOHN visualisation and human guidance to spot such patterns.

The possibility of adding a link function to a MOHN was proposed in section 3.7.1 but not pursued. Using a logit function would allow for binary classification, for example. The use of an exponential link function would make the MOHN equivalent to the Markov random fields used in DEUM. Other learning algorithms would be more appropriate than those studied here in these cases, iteratively reweighted least squares [63], for example is preferable to OLS when the output errors are not normally distributed.

Training a large MOHN is a suitable task for modern distributed computing techniques such as cluster computing or the use of a GPU. With a large data set distributed across a cluster, it is possible that a mapreduce job could build smaller local models in the map phase to produce a large ensemble that is then averaged in the reduce phase. Diversity across the ensemble could be managed by controlling the distribution of weight orders each map task samples from or by allowing some map tasks to explore while others exploit.

5.2 Conclusions

5.2.1 *Main Contribution*

Section 1.3 lists the properties we have claimed for a MOHN, the key algorithms required to build and search one, and some experimental findings. That list is summarised below with references to the parts of the thesis that hold a proof, algorithm or experimental evidence for the claims.

1. **Basis Function**: Section 3.2.2 provides a proof of equivalence between the Walsh basis and the MOHN basis. Section 3.5 show that basis coefficients can represent any function as a set of weak constraints. A MOHN can represent any function in $f : \{-1, 1\}^n \rightarrow \mathbb{R}$ as a MOHN function and a weak constraint set.

2. **Sparsity**: The MSDA, which is algorithm 7 on page 72 proposes a method for finding the non-zero weights for a MOHN. The approach is motivated by the need to work with small data sets. Experiments such as those in section 4.4 provide evidence that the MSDA can discover the correct sparse structure from fewer data points than other methods in the literature.

3. **Linear Parameter Models**: Equation 3.1 on page 53 shows the form of the MOHN linear parameter model. A MOHN has the well known properties of such a model. The experiment on page 151 demonstrates how a MOHN with the correct structure can learn randomly generated functions with a number of noise free training points equal in size to the number of weights in the MOHN. The mean squared error, lasso and ridge cost functions are convex for such models, which means there are no local minima in the error function, and removes a source of variance compared to non-convex functions such as the equivalents for an MLP. Illustrative examples are given on pages 111, 122 and 188.

4. **Interpretability**: The structure and values of the parameters in a MOHN have a meaning that is open to human interpretation. This allows networks to be visualised, compared and (to some extent) produce human readable facts. Section 3.6 proposes a network visualisation method and figure 4.31 on page 163 shows an annotated visualisation of a MOHN's weights and highlights its human readable nature. Section 3.6.1.1 explains how an ensemble of MOHNs can be combined into a single average model due to the fact that the weights play the same role in each model in an ensemble. There is an example of this being used in section 4.10.2.

Heuristic algorithms for performing two important tasks with MOHNs were presented:

1. **Structure Discovery**: The MSDA was proposed in section 3.3, as discussed above.

2. **Model Search**: Versions of local search algorithms designed to make use of the MOHN weight structure were proposed including random restart hill climb, iterated local search and variable neighbourhood search. Methods for performing simulated annealing and a method that allowed a MOHN to forget local optima were also proposed and tested. Different methods were found to suit some problems better than others, but all offered a gain in efficiency from only needing to partially evaluate the MOHN function on each step of the search. Examples of the MOHN structure leading a search algorithm directly to the global optimum when a simple hill climb would fail are given in sections 4.12 and 4.15. More work is needed to investigate how much can be gained from the MOHN structure during search, and how broadly any gains might apply.

The thesis also presents experimental evidence to suggest that MOHNs compare well with MLPs for regression modelling and EDAs for optimisation in certain circumstances.

1. **Non-Linear Regression**: For a number of benchmark functions, a MOHN showed advantages over an multi layer perceptron including finding a lower test error, requiring fewer data points, using fewer training epochs and displaying less variance across a number of training runs. Illustrative examples are given on pages 111, 122 and 188.

2. **Fitness Function Models**: MOHNs are capable of modelling benchmark fitness functions and finding the input values that produce the global maximum output of those functions in fewer evaluations of the fitness function than a number of published state-of-the-art methods. Section 4.11 provides examples where a MOHN is able to model and search a function in fewer evaluations than those reported in the literature for EDA methods using Markov random fields, Boltzmann machines, and Bayesian optimisation.

5.2.2 *Other Results*

A MOHN has also been shown to function as a high capacity content addressable memory, though at the expense of not being able to learn new patterns incrementally. An incremental approach that adds weights as it adds patterns may be possible, but that is left for future work. It was also shown that learning a Hamming distance based function where the desired memories are local maxima in the function reduces the number of spurious attractors. When the structure of the network is correct, the number of spurious attractors is reduced to zero. CAMs are probably of less interest in their own right but their study reveals points of interest about the capacity of regression models to capture multiple turning points in a function (and, consequently, local optima in a fitness function).

Most of the examples given in this work have involved variables that played a defined role in the function to be learned, for example Age in the examples using Experian data or the variable that encodes a certain node taking a certain colour in the graph colouring problem.

Signal processing type applications (image or speech recognition, for example) are more likely to benefit from the stages a deep neural network can offer: convolution and repeated feature extraction, for example. MOHNs are inherently shallow in their architecture but it might be interesting to investigate deep networks where some layers take the form of a MOHN.

The MOHN architecture might seem cumbersome compared to the more elegant multi layer perceptron. The necessity to perform structure discovery independently, rather than let the hidden units carry it out might seem like an unnecessary complication to an existing and well used technique. This criticism might gain strength when applied to real valued MOHNs that require added nodes for higher powers of the inputs too. Certainly there will be cases where an MLP out performs a MOHN, and times when a MOHN's ability to avoid local minima in the error function make it a better choice. In general machine learning involves trying a number of methods from a tool box and comparing their performance and the results presented in this thesis make a strong case for a place in that tool box for a MOHN.

My personal experience from machine learning with noisy data from commercial applications is that a simple linear model is rarely sufficient, but that the interactions between variables are generally of low order and the gains of an MLP over linear regression, while significant, are generally not massive. Such data is generally a mix of binary, nominal and real valued inputs and so a standard MOHN with a few real valued nodes would probably be a good choice. Certainly, I would include a MOHN in future consultancy jobs where the data was suitable (the Experian data, for example) as the insight they provide is so valuable.

BIBLIOGRAPHY

[1] David H. Ackley, Geoffrey E. Hinton, and Terrence J. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Science*, 9(1):147–169, 1985.

[2] Matthew Alden and Risto Miikkulainen. Marleda: Effective distribution estimation through markov random fields. Technical Report TR-13-18, Department of Computer Science, The University of Texas at Austin, Austin, TX, November 2013.

[3] Timothy L Andersen and Tony R Martinez. Dmp3: A dynamic multilayer perceptron construction algorithm. *International journal of neural systems*, 11(02):145–165, 2001.

[4] M.G Augasta and T. Kathirvalavakumar. Reverse engineering the neural networks for rule extraction in classification problems. *Neural Processing Letters*, 35(2):131–150, 2012.

[5] M.G. Augasta and T. Kathirvalavakumar. Rule extraction from neural networks - a comparative study. pages 404–408, 2012. cited By (since 1996)0.

[6] T. Bäck. A user's guide to genesys 1.0. Technical report, 1992.

[7] J. Bala, K. De Jong, J. Huang, H. Vafaie, and H. Wechsler. Using learning to facilitate the evolution of features for recognizing visual concepts. *Evolutionary Computation*, 4:297–311, 1996.

[8] Shumeet Baluja and Scott Davies. Combining multiple optimization runs with optimal dependency trees. Technical report, DTIC Carnegie Mellon University, 1997.

[9] Shumeet Baluja and Scott Davies. Using optimal dependency-trees for combinatorial optimization: Learning the structure of the search space. pages 30–38. Morgan Kaufmann, 1997.

[10] Eric B. Bartlett. Dynamic node architecture learning: An information theoretic approach. *Neural Networks*, 7(1):129–140, 1994.

[11] Elena Băutu, Sun Kim, Andrei Băutu, Henri Luchian, and Byoung-Tak Zhang. Evolving hypernetwork models of binary time series for forecasting price movements on stock markets. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 166–173. IEEE, 2009.

[12] K.G. Beauchamp. *Applications of Walsh and Related Functions*. Academic Press, London, 1984.

[13] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*, pages 437–478. Springer, 2012.

[14] Chris M Bishop. Training with noise is equivalent to tikhonov regularization. *Neural computation*, 7(1):108–116, 1995.

[15] E. K. Blum. Approximation of boolean functions by sigmoidal networks: Part i: Xor and other two-variable functions. *Neural Comput.*, 1(4):532–540, December 1989.

[16] Jeremy S. De Bonet, Charles L. Isbell, Jr., and Paul Viola. Mimic: Finding optima by estimating probability densities. In *Advances In Neural Information Processing Systems*, page 424. The MIT Press, 1996.

[17] Jakramate Bootkrajang, Sun Kim, and Byoung-Tak Zhang. Evolutionary hypernetwork classifiers for protein-protein interaction sentence filtering. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 185–192. ACM, 2009.

[18] Remco R. Bouckaert. Probalistic network construction using the minimum description length principle. In *ECSQARU*, pages 41–48, 1993.

[19] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[20] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, September 1973.

[21] A Brownlee. *Multivariate Markov Networks for Fitness Modelling in an Estimation of Distribution Algorithm*. PhD thesis, Robert Gordon University, 2009.

[22] Cristian Bucilua, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541. ACM, 2006.

[23] Erick Cantú-Paz. Feature subset selection with hybrids of filters and evolutionary algorithms. In *Scalable Optimization via Probabilistic Modeling*, pages 291–314. Springer, 2006.

[24] Gonzalo Joya Caparrós, Miguel A. Atencia Ruiz, and Francisco Sandoval Hernández. Hopfield neural networks for optimization: study of the different dynamics. *Neurocomputing*, 43(1-4):219–237, 2002.

[25] Pierre Chardaire, Jean Luc Lutton, and Alain Sutter. Thermostatistical persistency: A powerful improving concept for simulated annealing algorithms. *European Journal of Operational Research*, 86(3):565–579, 1995.

[26] Yutian Chen and Max Welling. Bayesian structure learning for markov random fields with a spike and slab prior. *arXiv preprint arXiv:1408.2047*, 2014.

[27] Daniel L Chester. Why two hidden layers are better than one. In *Proceedings of the international joint conference on neural networks*, volume 1, pages 265–268, 1990.

[28] Siddhartha Chib and Edward Greenberg. Understanding the Metropolis-Hastings Algorithm. *The American Statistician*, 49(4):327–335, 1995.

[29] Francisco Chicano, Darrell Whitley, and Andrew M Sutton. Efficient identification of improving moves in a ball for pseudo-boolean problems. In *Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 437–444. ACM, 2014.

[30] David Maxwell Chickering, David Heckerman, and Christopher Meek. Large-sample learning of bayesian networks is np-hard. *Journal of Machine Learning Research*, 5:1287–1330, 2004.

[31] Nicos Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, DTIC Document, 1976.

[32] David Coffin and Robert E Smith. Linkage learning in estimation of distribution algorithms. In *Linkage in evolutionary computation*, pages 141–156. Springer, 2008.

[33] Gregory F. Cooper and Edward Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9:309–347, 1992.

[34] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[35] Corinne Dahinden, Giovanni Parmigiani, Mark C. Emerick, and Peter Bühlmann. Penalized likelihood for sparse contingency tables with an application to full-length cdna libraries. *BMC Bioinformatics*, 8(1):1–11, 2007.

[36] George E Dahl, Dong Yu, Li Deng, and Alex 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.

[37] Yuval Davidor. Epistasis variance: A viewpoint on GA-hardness. In *Foundations of Genetic Algorithms*, pages 23–35, San Francisco, 1990. Morgan Kaufmann.

[38] Cassio P De Campos and Qiang Ji. Efficient structure learning of bayesian networks using constraints. *The Journal of Machine Learning Research*, 12:663–689, 2011.

[39] Stephen Della Pietra, Vincent Della Pietra, and John Lafferty. Inducing features of random fields. *IEEE transactions on pattern analysis and machine intelligence*, 19(4):380–393, 1997.

[40] K.L. Du and M.N.S. Swamy. *Neural Networks and Statistical Learning*. SpringerLink : Bücher. Springer London, 2013.

[41] Bradley Efron, Trevor Hastie, Iain Johnstone, Robert Tibshirani, et al. Least angle regression. *The Annals of statistics*, 32(2):407–499, 2004.

[42] Marcus Frean. The upstart algorithm: A method for constructing and training feedforward neural networks. *Neural computation*, 2(2):198–209, 1990.

[43] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of statistical software*, 33(1):1, 2010.

[44] Yasser Ganjisaffar. Lasso4j. https://github.com/yasserg/lasso4j.

[45] Nicolás García-Pedrajas, Domingo Ortiz-Boyer, and César Hervás-Martínez. An alternative approach for neural network evolution with a genetic algorithm: Crossover by combinatorial optimization. *Neural Networks*, 19(4):514–528, 2006.

[46] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.

[47] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

[48] Mark A. Hall. Correlation-based feature selection for discrete and numeric class machine learning. In *ICML*, pages 359–366, 2000.

[49] Leonard G.C. Hamey. {XOR} has no local minima: A case study in neural network error surface analysis. *Neural Networks*, 11(4):669 – 681, 1998.

[50] J.M Hammersley and P Clifford. Markov fields on finite graphs and lattices. *Unpublished*, 1971.

[51] Georges R. Harik, Fernando G. Lobo, and Kumara Sastry. Linkage learning via probabilistic modeling in the extended compact genetic algorithm (ecga). In *Scalable Optimization via Probabilistic Modeling*, pages 39–61. 2006.

[52] Trevor Hastie, Robert Tibshirani, Jerome Friedman, T Hastie, J Friedman, and R Tibshirani. *The elements of statistical learning*. Springer, 2009.

[53] Simon Haykin and Neural Network. A comprehensive foundation. *Neural Networks*, 2(2004):41, 2004.

[54] Robert B Heckendorn and Alden H Wright. Efficient linkage discovery by limited probing. *Evolutionary computation*, 12(4):517–545, 2004.

[55] David Heckerman, Dan Geiger, and David Maxwell Chickering. Learning bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20(3):197–243, 1995.

[56] Federico Heras, Javier Larrosa, and Albert Oliveras. Minimaxsat: An efficient weighted max-sat solver. *J. Artif. Intell. Res.(JAIR)*, 31:1–32, 2008.

[57] Geoffrey Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14:2002, 2000.

[58] Geoffrey Hinton. A practical guide to training restricted boltzmann machines. Technical report, University of Toronto, 2010.

[59] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554, 2006.

[60] R. R. Hocking. A biometrics invited paper. the analysis and selection of variables in linear regression. *Biometrics*, 32(1):pp. 1–49, 1976.

[61] Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[62] J.H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975.

[63] Paul W Holland and Roy E Welsch. Robust regression using iteratively reweighted least-squares. *Communications in Statistics-theory and Methods*, 6(9):813–827, 1977.

[64] Young-Seok Hong, Hungu Lee, and Min-Jea Tahk. Acceleration of the convergence speed of evolutionary algorithms using multi-layer neural networks. *Engineering Optimization*, 35(1):91–102, 2003.

[65] J. J. Hopfield and D. W. Tank. Neural computation of decisions in optimization problems. *Biological Cybernetics*, 52:141–152, 1985.

[66] John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences USA*, 79(8):2554–2558, April 1982.

[67] Eduardo R. Hruschka and Nelson F.F. Ebecken. Extracting rules from multilayer perceptrons in classification problems: A clustering-based approach. *Neurocomputing*, 70(1-3):384 – 397, 2006. Neural Networks Selected Papers from the 7th Brazilian Symposium on Neural Networks (SBRN '04) 7th Brazilian Symposium on Neural Networks.

[68] Hemant Ishwaran and J Sunil Rao. Spike and slab variable selection: frequentist and bayesian strategies. *Annals of Statistics*, pages 730–773, 2005.

[69] Tommi Jaakkola, David Sontag, Amir Globerson, and Marina Meila. Learning bayesian network structure using lp relaxations. In *International Conference on Artificial Intelligence and Statistics*, pages 358–365, 2010.

[70] Yaochu Jin, Michael Hüsken, Markus Olhofer, and Bernhard Sendhoff. Neural networks for fitness approximation in evolutionary optimization. In Yaochu Jin, editor, *Knowledge Incorporation in Evolutionary Computation*, volume 167 of *Studies in Fuzziness and Soft Computing*, pages 281–306. Springer Berlin Heidelberg, 2005.

[71] K. Jivani, J. Ambasana, and S Kanani. A survey on rule extraction approaches based techniques for data classification using neural network. *International Journal of Futuristic Trends in Engineering and Technology*, 1(1), 2014.

[72] H. Kargupta. The gene expression messy genetic algorithm. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 814–819, 1996.

[73] Hillol Kargupta and Kevin Buescher. The gene expression messy genetic algorithm for financial applications. In *Computational Intelligence for Financial Engineering, 1996., Proceedings of the IEEE/IAFE 1996 Conference on*, pages 155–161. IEEE, 1996.

[74] Hyun-Woo Kim, Byoung-Hee Kim, and Byoung-Tak Zhang. Evolutionary hypernetworks for learning to generate music from examples. In *Fuzzy Systems, 2009. FUZZ-IEEE 2009. IEEE International Conference on*, pages 47–52. IEEE, 2009.

[75] Joo-Kyung Kim and Byoung-Tak Zhang. Evolving hypernetworks for pattern classification. In *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pages 1856–1862. IEEE, 2007.

[76] Jason M. Kinser. Inability of higher-order outer product learning to map random higher-order problems. *Neurocomputing*, 8(3):349 – 357, 1995. Optimization and Combinatorics, Part I-III.

[77] Scott Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of statistical physics*, 34(5-6):975–986, 1984.

[78] Ron Kohavi and George H. John. Wrappers for feature subset selection. *Artif. Intell.*, 97(1-2):273–324, 1997.

[79] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[80] Anders Krogh and Jesper Vedelsby. Neural network ensembles, cross validation, and active learning. In *NIPS*, pages 231–238, 1994.

[81] T. Kubota. A higher order associative memory with Mcculloch-Pitts neurons and plastic synapses. In *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*, pages 1982 –1989, aug. 2007.

[82] Pedro Larrañaga, Cindy M. H. Kuijpers, Roberto H. Murga, and Yosu Yurramendi. Learning bayesian network structures by searching for the best ordering with genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 26(4):487–493, 1996.

[83] Saskia Le Cessie and Johannes C Van Houwelingen. Ridge estimators in logistic regression. *Applied statistics*, pages 191–201, 1992.

[84] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[85] Yann LeCun, John S Denker, Sara A Solla, Richard E Howard, and Lawrence D Jackel. Optimal brain damage. In *NIPS 89*, San Francisco, 1989. Morgan Kaufmann.

[86] Beom-Jin Lee, Jung-Wo Ha, Kyung-Min Kim, and Byoung-Tak Zhang. Evolutionary concept learning from cartoon videos by multimodal hypernetworks. In *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 1186–1192. IEEE, 2013.

[87] Su-In Lee, Varun Ganapathi, and Daphne Koller. Efficient structure learning of markov networks using *l*_1-regularization. In *Advances in neural Information processing systems*, pages 817–824, San Francisco, 2006. Morgan Kaufmann.

[88] Xuesong Li and Lin Ma. Minimizing binary functions with simulated annealing algorithm with applications to binary tomography. *Computer Physics Communications*, 183(2):309 – 315, 2012.

[89] Thomas Liddle. Kick strength and online sampling for iterated local search. In *Proceedings of the 45th Annual Conference of the ORSNZ*, 2010.

[90] Helena R Lourenço, Olivier C Martin, and Thomas Stützle. *Iterated local search*. Springer, 2003.

[91] Sean Luke. *Essentials of metaheuristics*. Lulu Com, 2013.

[92] David J.C. MacKay. Bayesian interpolation. *Neural Computation*, 4:415–447, 1991.

[93] Aravindh Mahendran and Andrea Vedaldi. Understanding deep image representations by inverting them. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5188–5196, 2015.

[94] Luigi Malago, Matteo Matteucci, and Gabriele Valentini. Introducing the l1-regularized logistic regression in markov networks based edas. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 1581–1588. IEEE, 2011.

[95] R. McEliece, E. Posner, E. Rodemich, and S. Venkatesh. The capacity of the hopfield associative memory. *Information Theory, IEEE Transactions on*, 33(4):461 – 482, jul 1987.

[96] Nicolai Meinshausen and Peter Bühlmann. High-dimensional graphs and variable selection with the lasso. *The annals of statistics*, pages 1436–1462, 2006.

[97] Toby J Mitchell and John J Beauchamp. Bayesian variable selection in linear regression. *Journal of the American Statistical Association*, 83(404):1023–1032, 1988.

[98] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers and Operations Research*, 24(11):1097 – 1100, 1997.

[99] Radford M Neal. Annealed importance sampling. *Statistics and Computing*, 11(2):125–139, 2001.

[100] Radford M Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.

[101] Pekka Parviainen and Mikko Koivisto. Exact structure discovery in bayesian networks with less space. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 436–443. AUAI Press, 2009.

[102] M. Pelikan, D.E. Goldberg, and E. Cantú-Paz. Linkage problem, distribution estimation, and bayesian networks. *Evolutionary computation*, 8(3):311–340, 2000.

[103] M. Pelikan and H. Mühlenbein. The bivariate marginal distribution algorithm. In R. Roy, T. Furuhashi, and P. K. Chawdhry, editors, *Advances in Soft Computing - Engineering Design and Manufacturing*, pages 521–535, London, 1999. Springer-Verlag.

[104] Martin Pelikan and David. E. Goldberg. Hierarchical bayesian optimization algorithm = Bayesian optimization algorithm + niching + local structures. pages 525–532. Morgan Kaufmann, 2001.

[105] Hanchuan Peng, Fuhui Long, and Chris H. Q. Ding. Feature selection based on mutual information: Criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(8):1226–1238, 2005.

[106] Simon Perkins, Kevin Lacker, and James Theiler. Grafting: Fast, incremental feature selection by gradient descent in function space. *Journal of machine learning research*, 3(Mar):1333–1356, 2003.

[107] Lutz Prechelt. Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer, 1998.

[108] Malte Probst and Franz Rothlauf. Deep boltzmann machines in estimation of distribution algorithms for combinatorial optimization. *arXiv preprint arXiv:1509.06535*, 2015.

[109] Malte Probst, Franz Rothlauf, and Jörn Grahl. Scalability of using restricted boltzmann machines for combinatorial optimization. *arXiv preprint arXiv:1411.7542*, 2014.

[110] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Advanced lectures on machine learning*, pages 63–71. Springer, 2004.

[111] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1. chapter Learning Internal Representations by Error Propagation, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[112] M. P. Vecchi S. Kirkpatrick, C. D. Gelatt. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[113] E.W. Saad and D.C. Wunsch II. Neural network explanation using inversion. *Neural Networks*, 20(1):78–93, 2007. cited By (since 1996)22.

[114] T. Samad and P. Harper. High-order hopfield and tank optimization networks. *Parallel Computing*, 16(2-3):287–292, 1990.

[115] Mark Schmidt. *Graphical Model Structure Learning with ℓ_1 Regularization*. PhD thesis, University of British Columbia, 2010.

[116] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[117] S. Shakya, A. Brownlee, J. McCall, F. Fournier, and G. Owusu. A fully multivariate deum algorithm. In *Evolutionary Computation, 2009. CEC '09. IEEE Congress on*, pages 479–486, 2009.

[118] Yi Shen, Xiaojun Zong, and Minghui Jiang. High-order hopfield neural networks. In Jun Wang, Xiaofeng Liao, and Zhang Yi, editors, *Advances in Neural Networks - ISNN 2005*, volume 3496 of *Lecture Notes in Computer Science*, pages 235–240. Springer Berlin Heidelberg, 2005.

[119] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. *CoRR*, abs/1312.6034, 2013.

[120] Jung-Woo Ha Soo-Jin Kim and Byoung-Tak Zhang. Bayesian evolutionary hypergraph learning for predicting cancer clinical outcomes. *J. of Biomedical Informatics*, 49(C):101–111, June 2014.

[121] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

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

[123] Matthew J Streeter. Upper bounds on the time and space complexity of optimizing additively separable functions. In *Genetic and Evolutionary Computation Conference*, pages 186–197. Springer, 2004.

[124] Joe Suzuki. Learning bayesian belief networks based on the MDL principle: An efficient algorithm using the branch and bound technique, 1998.

[125] Kevin Swingler. On the capacity of Hopfield neural networks as EDAs for solving combinatorial optimisation problems. In *Proc. IJCCI (ECTA)*, pages 152–157. SciTePress, 2012.

[126] Kevin Swingler. A walsh analysis of multilayer perceptron function. In *NCTA 2014 - Proceedings of the International Conference on Neural Computation Theory and Applications, part of IJCCI 2014, Rome, Italy, 22 - 24 October, 2014*, pages 5–14, 2014.

[127] Kevin Swingler. A comparison of learning rules for mixed order hyper networks. In *Proceedings of the 7th International Joint Conference on Computational Intelligence (IJCCI 2015) - Volume 3: NCTA, Lisbon, Portugal, November 12-14, 2015.*, pages 17–27, Setubal, 2015. ScITePress.

[128] Kevin Swingler. Local optima suppression search in mixed order hyper networks. In *Proc. UKCI 2015*, 2015.

[129] Kevin Swingler. Opening the black box: Analysing MLP functionality using Walsh functions. In Juan Julian Merelo, Agostinho Rosa, José M. Cadenas, António Dourado, Kurosh Madani, and Joaquim Filipe, editors, *Computational Intelligence*, volume 620 of *Studies in Computational Intelligence*, pages 303–323. Springer International Publishing, 2016.

[130] Kevin Swingler. Structure discovery in mixed order hyper networks. *Big Data Analytics*, 2016.

[131] Kevin Swingler. High capacity content addressable memory with mixed order hyper networks. In Juan Julian Merelo, Agostinho Rosa, José M. Cadenas, António Dourado, Kurosh

Madani, and Joaquim Filipe, editors, *Computational Intelligence*, Studies in Computational Intelligence. Springer International Publishing, in press.

[132] Kevin Swingler and Leslie S. Smith. Mixed order associative networks for function approximation, optimisation and sampling. In *ESANN 2013, 21st European Symposium on Artificial Neural Networks, Proceedings*, 2013.

[133] Kevin Swingler and Leslie S. Smith. An analysis of the local optima storage capacity of hopfield network based fitness function models. *Transactions on Computational Collective Intelligence XVII, LNCS*, 8790:248–271, 2014.

[134] Kevin Swingler and Leslie S. Smith. Training and making calculations with mixed order hyper-networks. *Neurocomputing*, (141):65–75, 2014.

[135] Yoel Tenne and Chi-Keong Goh. *Computational intelligence in expensive optimization problems*, volume 2. Springer Science & Business Media, 2010.

[136] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58:267–288, 1996.

[137] Renato Tintos, Darrell Whitley, and Francisco Chicano. Partition crossover for pseudo-boolean optimization. In *Proceedings of the 2015 ACM Conference on Foundations of Genetic Algorithms XIII*, FOGA '15, pages 137–149, New York, NY, USA, 2015. ACM.

[138] Miwako Tsuji, Masaharu Munetomo, and Kiyoshi Akama. Modeling dependencies of loci with string classification according to fitness differences. In *Genetic and Evolutionary Computation–GECCO 2004*, pages 246–257. Springer, 2004.

[139] Gabriele Valentini, Luigi Malagò, and Matteo Matteucci. Optimization by l1-constrained markov fitness modelling. In *Learning and Intelligent Optimization*, pages 250–264. Springer, 2012.

[140] Santosh S Venkatesh and Pierre Baldi. Programmed interactions in higher-order neural networks: Maximal capacity. *Journal of Complexity*, 7(3):316–337, 1991.

[141] Santosh S Venkatesht and Pierre Baldi. Programmed interactions in higher-order neural networks: The outer-product algorithm. *Journal of Complexity*, 7(4):443 – 479, 1991.

[142] J.L. Walsh. A closed set of normal orthogonal functions. *Amer. J. Math*, 45:5–24, 1923.

[143] Darrell Whitley and Wenxiang Chen. Constant time steepest descent local search with lookahead for nk-landscapes and max-ksat. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 1357–1364. ACM, 2012.

[144] L. Willmes, T. Back, Yaochu Jin, and B. Sendhoff. Comparing neural networks and kriging for fitness approximation in evolutionary optimization. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 1, pages 663 – 670 Vol.1, dec. 2003.

[145] G. V. Wilson and G. S. Pawley. On the stability of the travelling salesman problem algorithm of hopfield and tank. *Biol. Cybern.*, 58(1):63–70, January 1988.

[146] David H Wolpert and William G Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, 1997.

[147] Kok Sung Won, T. Ray, and Kang Tai. A framework for optimization using approximate functions. In *Evolutionary Computation, 2003. CEC '03. The 2003 Congress on*, volume 3, pages 1520–1527 Vol.3, 2003.

[148] Man Leung Wong, Wai Lam, and Kwong-Sak Leung. Using evolutionary programming and minimum description length principle for data mining of bayesian networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21(2):174–178, 1999.

[149] Man Leung Wong, Shing Yan Lee, and Kwong-Sak Leung. A hybrid data mining approach to discover bayesian networks using evolutionary programming. In *GECCO*, pages 214–222, 2002.

[150] Xin Yao and Yong Liu. Towards designing artificial neural networks by evolution. *Applied Mathematics and Computation*, 91(1):83–90, 1998.

[151] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *Computer vision–ECCV 2014*, pages 818–833. Springer, 2014.

[152] Tong Zhang. Solving large scale linear prediction problems using stochastic gradient descent algorithms. In *ICML 2004: Proceedings of the twenty first International Conference on Machine Learning.*, pages 919–926. Omnipress, 2004.