A Comparison of Learning Rules for Mixed Order Hyper Networks

Kevin Swingler
Computing Science and Maths, University of Stirling, FK9 4LA Stirling, Scotland, U.K.

Keywords: High Order Networks, Learning Rules.

Abstract: A mixed order hyper network (MOHN) is a neural network in which weights can connect any number of neurons, rather than the usual two. MOHNs can be used as content addressable memories with higher capacity than standard Hopfield networks. They can also be used for regression, clustering, classification, and as fitness models for use in heuristic optimisation. This paper presents a set of methods for estimating the values of the weights in a MOHN from training data. The different methods are compared to each other and to a standard MLP trained by back propagation and found to be faster to train than the MLP and more reliable as the error function does not contain local minima.

1 INTRODUCTION

For a long time, the multi layer perceptron (MLP) has been a very popular choice for performing non-linear regression on functions of multiple inputs. It has the advantage of being easy to apply to problems where there is little or no knowledge of the structure of the function to be learned, particularly concerning the interactions between input variables. The weight learning algorithm (back propagation of error, for example) simultaneously discovers features (interactions between inputs) in the function that underlies the training data and the correct values for the regression coefficients, given those features. This leads to two significant and well known disadvantages of the MLP, namely the so called 'black box problem' that means it is very difficult for a human analyst to learn much about the structure of the underlying function from the structure of the network and the problem of local minima in the error function that are a result of the hidden units failing to encode the correct interactions between input variables.

These problems are addressed by Mixed Order Hyper Networks (MOHNs) (Swingler and Smith, 2014a), which make the structure of the function explicit, meaning that human readability is greatly improved and there are no local minima in the error function. This paper presents and compares a number of methods for calculating the correct weight values for a MOHN of fixed structure. Different learning rules have different strengths and weaknesses. Some, for example may be carried out in an on line mode, meaning that the data need not be all stored in memory at one time. On line learning also allows partially learned networks to be updated in light of new data or as part of an algorithm to discover the correct connection structure. Standard regression techniques such as ordinary least squares (OLS) and Least Absolute Shrinkage and Selection Operator (LASSO) may be applied when on line learning is not required. This has the added advantage that confidence intervals may be calculated for the network weights. LASSO also has the advantage that weights that do not contribute to the function end up with values equal to zero.

MOHNs have been shown to be useful as fitness function models if used as part of a metaheuristic constraint satisfaction (or combinatorial optimisation) algorithm (Swingler and Smith, 2014b). In such cases, it is not always necessary to learn the whole function space correctly, but sufficient to build a model where the attractors in the energy function correspond to turning points (local optima) in the fitness function. A simpler learning rule is sufficient to build such models, and is presented in this paper.

2 MIXED ORDER HYPER NETWORKS

A Mixed Order Hyper Network is a neural network in which weights can connect any number of neurons. A MOHN has a fixed number of n neurons and $\leq 2^n$ weights, which may be added or removed dynamically during learning. Each neuron can be in one of three states: $u_i \in \{-1, 1, 0\}$ where a value of 0 indicates a wild card or unknown value. The state of the MOHN is determined by the values of the vector, $\mathbf{u} = u_0 \dots u_{n-1}$. The structure of a MOHN is defined by a set, $\mathbf{W}$ of real valued weights, each connecting $0 \leq k \geq n$ neurons. The weights define a hyper graph connecting the elements of $\mathbf{u}$. Each weight, w_j has an integer index that is determined by the indices of the neurons it connects:

$$\mathbf{W} \subseteq \{w_j : j = 0 \dots 2^{n-1}\} \quad w_j \in \mathbb{R} \tag{1}$$

The weights each have an associated order, defined by the number of neurons they connect. There is a single zero-order weight, which connects no neurons, but has a weight all the same. There are n first order weights, which are the equivalent of bias inputs in a standard neural network. In general, there are $\binom{n}{k}$ possible weights of order k in a network of size n. For convenience of notation, the set of k neurons connected to weight w_j is denoted Q_j, meaning that the index j defines a neuron subset. This is done by creating an n bit binary number, where bit i is set to one to indicate that neuron i is part of the subset and zero otherwise. The resulting binary number, stated in base 10 becomes the weight index. For example, the weight connecting neurons $\{0, 1, 2\}$ is w_7 as setting the bits 0,1,2 in a four bit number gives 0111, which is 7 in base 10. Consequently, we can write $Q_7 = \{u_0, u_1, u_2\}$. Figure 1 shows an example MOHN where $n = 4$.

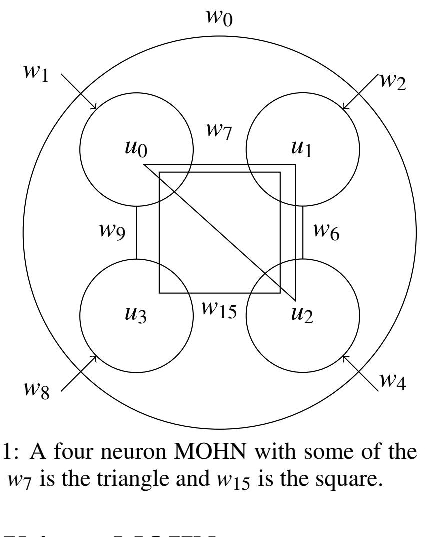

Figure 1: A four neuron MOHN with some of the weights shown. w_7 is the triangle and w_{15} is the square.

2.1 Using a MOHN

MOHNs can be applied to a number of different computational intelligence tasks such as building a content addressable memory, performing regression, clustering, classification, probability distribution estimation and as fitness function models for use in heuristic optimisation. These different tasks involve different methods of use and require different approaches to estimating the values on the weights.

3 THE WEIGHT ESTIMATION RULES

This section presents the different methods for estimating the weights needed to allow a MOHN to perform a particular task. Although, in theory the weights can be designed by hand, all of the methods described here are based on learning from data. In what follows, a single training example consists of a vector of input variables and a real valued output denoted $(\mathbf{x}, y)$. The training data as a whole is denoted D.

3.1 Hebbian Learning

The simplest of the MOHN learning rules is an extension to the Hebbian rule employed by a Hopfield network to allow it to work with higher order weights. In this case, the training data consists only of input patterns, $\mathbf{x}$ and no function output is specified. Learning involves setting $u_i = x_i$ for each neuron and the weight update is then:

$$w_j = w_j + \prod_{u \in Q_j} u \tag{2}$$

The Hebbian learning rule allows the MOHN to be used as a content addressable memory (CAM). The CAM learning algorithm is given in algorithm 1.

Algorithm 1: Loading Pattern **x** into a MOHN CAM.

```
$u_i = x_i \forall i$          ▷ Set the neuron outputs to equal the pattern to be learned
$w_j = w_j + \prod_{u \in Q_j} u \; \forall w_j \in W$   ▷ Update the weights according to equation 2
```

For a network that is fully connected at order two, algorithm 1 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. 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 a neuron update rule that first calculates an activation value for each neuron, a_i using equation 3 and then applies a threshold using equation 4.

$$a_i = \sum_{j : u_i \in Q_j} \left(w_j \prod_{k \in Q_j \setminus i} u_k \right) \tag{3}$$

where $j : u_i \in Q_j$ makes j enumerate the index of each weight that connects to u_i and $k \in Q_j \setminus i$ indicates the index of every member of Q_j, except neuron i itself. A neuron's output is then calculated using the threshold function in equation 4.

$$u_i = \begin{cases} 1, & \text{if } a_i > 0 \\ -1, & \text{otherwise} \end{cases} \tag{4}$$

An attractor state is a pattern across **u** from which the application of equation 3 results in no change to any of the neuron outputs. A trained MOHN settles to an attractor point by repeated application of the activation rules 3 and 4, choosing neurons in random order. Algorithm 2 describes the algorithm for settling from a pattern to an attractor:

Algorithm 2: Settling a trained MOHN to an attractor point.

```
repeat
    $ch \leftarrow FALSE$  ▷ Keep track of whether or not a change has been made
    $visited \leftarrow \{\}$  ▷ Keep track of which neurons have been visited
    repeat
        $i \leftarrow rand(i : i \notin visited)$  ▷ Pick a random unset neuron
        $temp \leftarrow u_i$  ▷ Make a note of its value for later comparison
        Update$(u_i)$  ▷ Update the neuron's output using equations 3 and 4
        if $u_i \neq temp$ then
            $ch \leftarrow TRUE$
        end if  ▷ If a change was made to the neuron's output, note the fact
        $visited \leftarrow \{visited \cup i\}$  ▷ Add the neuron's index to the visited set
    until $\|visited\| = n$  ▷ Loop until all neurons have been updated
until $ch = FALSE$  ▷ Loop if any neuron value has changed
```

The dynamics of algorithm 2 have an underlying Lyapunov function, just as they do in a standard HNN and will always settle to a local minimum of the associated energy function. (Venkatesh and Baldi, 1991) report a capacity for binary valued order k networks of the order of $n^k/\ln n$, a figure that is also reported by (Kubota, 2007). In fact, as described below, such networks are capable of representing any arbitrary Lyaponov function and therefore a network with the right structure will be able to represent any possible number of turning points.

3.2 Weighted Hebbian Learning

Let $f(\mathbf{x})$ be a multi-modal function where each local maximum represents a pattern of interest. These patterns might be local optima in an optimisation task, archetypes in a clustering task or examples of a satisfaction of multiple constraints, for example. A MOHN can be trained as a CAM in which the attractors are the local maxima of the function. The learning rule is a weighted version of the Hebbian rule:

$$w_j = \sum_{\mathbf{x} \in D} \frac{1}{|D|} f(\mathbf{x}) \prod_{u \in Q_i} u \tag{5}$$

Previous work (Swingler and Smith, 2014b) has shown that the weighted Hebbian rule is capable of learning the local maxima of a function from samples of $\mathbf{x}, f(\mathbf{x})$ and that the capacity of the resulting networks for storing such attractors was equal to the capacity of a CAM trained using equation 2. The difference between equations 2 and 5 is that the target patterns are known in the first case, but unknown in the second, where they are local maxima of y in a function that is learned from a sample of $(\mathbf{x}, y)$ pairs. Note also that experiments have shown that the training data need not contain a single example of any of the attractor patterns for the method to work.

3.2.1 Parity Count Learning

When the inputs (both single variables and products of variable subsets) are uncorrelated (i.e. orthogonal) and each input has an even distribution of values, the weighted Hebbian rule produces the correct weight values in a single pass of the data. When the distribution of values across each variable is uneven, a better estimate of weight values may be made by taking into account how often the input product on each weight is positive or negative during learning. Each weight is set to equal the difference between the average of the output y when the weight's input is positive and when it is negative.

Let D_j^+ be the set of sub-patterns learned by w_j that contain values whose product is positive and D_j^- be the set of sub-patterns learned by w_j that contain values whose product is negative. Now let $\langle y^+ \rangle$ be the average value of y associated with the members of D_j^+ and $\langle y^- \rangle$ be the average value of y associated with the members of D_j^-:

$$\langle y^+ \rangle = \frac{1}{|D_j^+|} \sum_{\mathbf{x} \in D_j^+} f(\mathbf{x}) \tag{6}$$

Similarly, $\langle y^- \rangle$ is calculated as a sum over $\mathbf{x} \in D_j^-$. The weight calculation is simply

$$w_j = \frac{1}{2}\left(\langle y^+\rangle - \langle y^-\rangle\right) \tag{7}$$

The averages may be maintained online so that the weight values are always correct at any time during learning (rather than summing and dividing at the end of a defined training set). W_0 is set in a similar way. The Weighted Hebbian calculation, $w_0 = \langle y\rangle$ means that w_0 is just the average of the output, y across the training sample. This can be improved by taking into account the distribution of patterns across each input.

$$w_0 = \langle y\rangle - \sum_{j:w_j\in W} w_j \Big\langle \prod_{x\in w_j} x\Big\rangle \tag{8}$$

where $x \in w_j$ indicates the neurons connected to weight w_j and $\left\langle \prod_{x\in w_j} x\right\rangle$ is the average value across all input patterns of the product of the values of x connected to w_j.

3.3 Regression Rules

The weighted Hebbian update rule is capable of capturing the turning points in a function, but cannot accurately reproduce the output of the function itself across all of the input space. Such networks have an energy function[1] and this can be used as a regression function for estimating $\hat{y} = f(\mathbf{x})$ in the form

$$\hat{y} = \sum_i w_j \prod_{u\in Q_i} u \tag{9}$$

The weight values for the regression may be calculated either in a single off line calculation or using an on line weight update rule.

3.3.1 Off Line Regression

To use ordinary least squares (OLS) (Hastie et al., 2009) to estimate the weights offline, a matrix X is constructed where each row represents a training example and each column represents a weight. The first column represents W_0 and always contains a 1. The remaining columns contain the product of the values of the inputs connected by the column's weight, $\prod_{x\in Q_i} x$. A vector Y takes the output values associated with each of the input rows and the parameters are calculated using singular value decomposition:

$$\beta = (X^T X)^{-1} X^T y \tag{10}$$

where X^T is the transpose of X, X^{-1} is the inverse of X and β becomes a vector from which the weights of the MOHN may be directly read so that $w_0 = \beta_0$ and the remaining weights take values from β in the same sequence as they were inserted into the matrix X.

[1]The regression equation 9 is actually the negative of the energy function, which is minimised by applying the settling algorithm 2.

3.3.2 LASSO Learning Rule

The LASSO algorithm (Tibshirani, 1996) may also be used to learn the 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 LASSO are read back into the weights of the MOHN in the same order. LASSO performs regression with an additional constraint on the L^1 norm of the weight vector. The learning algorithm minimises the sum:

$$\sum_{\mathbf{x}\in D} (f(\mathbf{x}) - \hat{f}(\mathbf{x})) + \lambda \sum_{w\in W} |w| \tag{11}$$

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 take a value of zero. This not only allows LASSO to reject input variables that contribute little, but also to reject higher order weights that are not needed. 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 LASSO regression has been performed.

3.3.3 On Line Learning

The weights of a MOHN can also be estimated on line (where the data is streamed one pattern at a time, rather than being available in a matrix as in equation 10) using a linear version of the delta learning rule, the Linear Delta Rule (LDR):

$$w_i = w_i + \alpha(f(\mathbf{x}) - \hat{f}(\mathbf{x})) \prod_{u\in Q_i} u \tag{12}$$

where $\alpha < 1$ is the learning rate. Experimental results have suggested that one divided by the number of weights in the network is a good value for α, i.e $\alpha = \frac{1}{|W|}$. This allows the correction made in response to each prediction error to be spread across all of the weights.

The online learning algorithm is very similar to the perceptron (or MLP) learning algorithm. The iterative nature of the algorithm allows for early stopping to be used to control for overfitting with reference to an independent test set. Algorithm 3 describes the learning process.

Algorithm 3: On Line MOHN Learning with the Linear Delta Rule.

```
Let $D_r$ be a subset of the available data to be used for training the network
Let $D_s$ be a subset of the available data to be used for testing the network
for all $(\mathbf{x}, f(\mathbf{x})) \in D_r$ do
    Initialise the weights in the network using the parity rule of equation 7
end for
repeat
    for all $\mathbf{x} \in D_r$ do
        Update the weights in the network using the delta learning rule of equation 12
        Let $e$ be the root mean squared error that results from evaluating every member of $D_s$ with the model
    end for
until $e$ is sufficiently low or starts to increase consistently
```

Note that the weights are initialised with the parity count learning rule, 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 imeadiately, without the need for additional error descent learning. The learning algorithm will work without the initialisation (the weights can be set to zero) but then requires more iterations of the learning cycle.

4 ANALYSIS OF LEARNING RULES

This section begins with a summary of the abilities and limitations of the different learning rules presented in this paper. It then goes on to analyse the rules and the resulting networks. Table 1 summarises some of the differences between the methods. Due to the structure of the MOHN, all of the learning rules are capable of reproducing the maximal turning points of the learned function, but the Hebbian based rules do not minimise the error elsewhere in the function space. The Hebbian rules learn in a single presentation of the data, so can operate in an on line mode without the need to iterate through the data set more than once. The others require either on line iterations or the entire data set to be present off line.

The weighted Hebbian rule is accurate only when a full sample of the input/output space is available, so is of limited practical use as the parity counting method gives more accurate estimates operating on weights independently with a single pass through the data. The parity counting method provides good starting weights for the linear delta rule.

Table 1: Comparing four different MOHN learning rules in terms of the learning mode, any regularisation that is possible, whether or not the training error is minimised, and whether the training data is presented as input,output pairs (IO) or as patterns to store in a content addressable memory (CAM).

Method	Mode	Reg.	Min. Err.	Data
Hebbian	One shot	None	No	CAM
Weighted Hebb	One shot	None	No	IO
LDR	On line	Early Stop	Yes	IO
OLS	Off line	None	Yes	IO
LASSO	Off line	L^1-norm	Yes	IO

The following sections investigate different network structures in more detail.

4.1 Second Order Networks

When a MOHN has only second order connections, it is equivalent to a Hopfield Neural Network (HNN) (Hopfield, 1982) and the Hebbian learning rule of equation 2 is the standard learning rule for a HNN. It is well known that HNNs are able to learn patterns as content addressable memories, but that they suffer from the presence of spurious attractors too. These spurious attractors may be removed (or their presence avoided) by defining an energy function for the network in which the patterns to be stored as memories are local maxima. This may be done using a Hamming distance based function and (Swingler, 2012), (Swingler and Smith, 2014b) have shown that using the weighted Hebbian update rule of equation 5 and such a function on a second order MOHN (or equivalently, a HNN) is sufficient to produce a content addressable memory in which the turning points of the function are the memories to be stored. The capacity of a HNN for storing patterns is the same if the patterns are loaded directly with the Hebbian learning rule as it is when the patterns are learned from a Hamming distance based function.

To build the Hamming distance based function, denote the set of patterns to be stored as $\mathbf{T}$:

$$\mathbf{T} = \{t_1, \ldots, t_s\} \tag{13}$$

and define a set of sub-functions, $f(\mathbf{x}|t_j)$ as a weighted Hamming distance between $\mathbf{x}$ and each target pattern t_j in $\mathbf{T}$ as

$$f(\mathbf{x}|t_j) = \sum \frac{\delta_{x_i, t_{ji}}}{n} \tag{14}$$

where t_{ji} is element i of target j and $\delta_{x_j,t_{ji}}$ is the Kronecker delta function between pattern element i in t_j and its equivalent in $\mathbf{x}$. The function output given an input pattern, $f(\mathbf{x})$ is the maximal output across all the sub-functions given an input of $f(\mathbf{x})$.

$$f(\mathbf{x}|\mathbf{t}) = max_{j=1...s}(f(\mathbf{x}|t_j)) \tag{15}$$

By generating random input patterns, evaluating each using equation 15 and then using the LDR of equation 12 to learn each input,output pair sampled, a network with attractors at each member of $\mathbf{T}$ is learned. The network has the additional quality that as the number of samples learned increases, the number of spurious attractors in the network decreases.

To test this claim, experiments were run in which a 100 neuron MOHN was trained on a function that contained four true attractor states. Figure 2 shows the average results of running 100 trials in which the number of spurious attractors and the error of the network were measured for each iteration of the training data, which was a random sample of size 20,000 from the Hamming distance based function 15. In each case, the number of spurious attractors was reduced to zero as the training error approached zero.

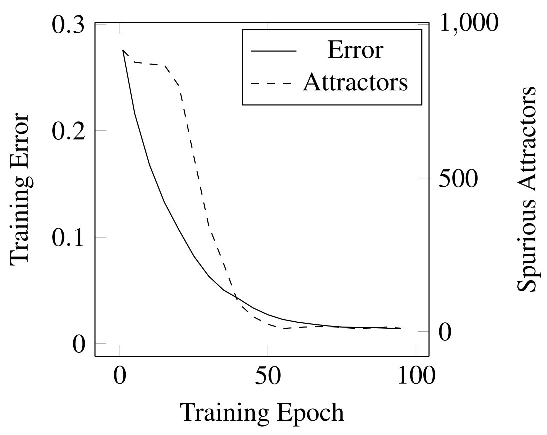

Figure 2: As the number of learning iterations increases, the training error decreases as does the number of spurious attractors in the model.

Researchers have shown how the weights of a HNN can be designed to represent the travelling salesman problem (Hopfield and Tank, 1985), (Wilson and Pawley, 1988) and other problems such as graph colouring (Caparrós et al., 2002). These approaches are limited by the fact that the weights must be chosen by hand to reflect the constraints of the problem to be solved. By training a HNN (or a MOHN) by sampling from a fitness function, it is now possible to build a network to represent any problem with a fitness function that can be evaluated, not just those that are amenable to having their weights set by hand.

4.2 Full Networks

When the data are noise free, a network is fully connected and the data sample is exhaustive (i.e. it covers every possible input pattern once), the weighted Hebbian rule of equation 5 (with $|D| = 2^n$) will produce weights which reproduce the target function perfectly. In such cases, the product $\prod_{u \in Q_i} u$ provides a basis function for $f : \{-1,1\}^n \to \mathbb{R}$. This basis function is very similar to the well know Walsh basis (Walsh, 1923), (Beauchamp, 1984).

A Walsh representation of a function $f(\mathbf{x})$ is defined by a vector of parameters, the Walsh coefficients, $\omega = \omega_0 \dots \omega_{2^n-1}$. Each ω_j is associated with the Walsh function ψ_j. The Walsh representation of $f(\mathbf{x})$ is constructed as a sum over all ω_j. In the sum, each ω_j is either added to or subtracted from the total, depending on the value of the Walsh function $\psi_j(x)$ which gives the function for the Walsh sum:

$$f(\mathbf{x}) = \sum_{j=0}^{2^{n-1}} \omega_j \psi_j(x) \tag{16}$$

A Walsh function, $\psi_j(x)$ returns +1 or -1 depending on the parity of the number of 1 bits in shared positions across $\mathbf{x}$ and $\mathbf{j}$ where $\mathbf{j}$ is the binary representation of the integer j. Using logical notation, a Walsh function is derived from the result of an XOR (parity count) of an AND (agreement of bits with a value of 1):

$$\psi_j(x) = \oplus_{i=1}^{n}(x_i \wedge j_i) \tag{17}$$

where $\oplus$ is a parity operator, which returns 1 if the argument list contains an even number of 1s and -1 otherwise. The Walsh transform of an n-bit function, $f(\mathbf{x})$, produces 2^n Walsh coefficients, ω_j, indexed by the 2^n combinations across $f(\mathbf{x})$. Each Walsh coefficient, ω_j is calculated by

$$\omega_j = \frac{1}{2^n} \sum_{x=0}^{2^n-1} f(x)\psi_j(x) \tag{18}$$

The weight values in a fully trained MOHN are equal in magnitude to the Walsh coefficients of the same index, but that they differ in sign when the weight order is an odd number. That is,

$$\omega_j = p(\omega_j) w_j \quad \forall w_j \in W \tag{19}$$

where $p(\omega_j)$ is the parity of the order of ω_j such that:

$$p(\omega_j) = \begin{cases} 1 & \text{if the order of } i \text{ is even} \\ -1 & \text{otherwise} \end{cases} \tag{20}$$

This is because the Walsh function returns a value based on a parity count of the number of variables set to one across the input variables that are connected to a given coefficient, as shown in equation 17. The parity function returns 1 if the number of variables with a value of one is even and -1 otherwise. The MOHN uses the product of those same values, which evaluates to -1 whenever there is an odd number of inputs set to -1. The MOHN indices match the Walsh coefficient indices because they both use the same method of deriving the index number from the binary representation of the connections described in section 2.

As a fully connected MOHN provides a basis for all possible functions in $f : \{-1,1\}^n \to \mathbb{R}$, then it follows that any function with coefficient values of zero may be perfectly represented by a less than fully connected MOHN so providing the correct structure can be found, a MOHN may represent any arbitrary function.

4.3 Discovering Network Structure

The structure of a MOHN is defined by W, which is a subset of all possible 2^n weights. As noted above, a fully connected second order network implements a HNN and a fully connected network at all orders forms a basis of all functions $f : \{-1,1\}^n \to \mathbb{R}$. Any other pattern of connectivity is also possible, for example a first order only network is equivalent to a perceptron, or a multiple linear regression model. Adding higher order weights increases the power of the model to represent more complex functions.

Discovering the correct structure for the network is both challenging and instructive, compared to the same task when using an MLP, which is quite straight forward, but done in the dark. The question of discovering structure in functions from samples of data is of particular importance in the field of metaheuristic optimisation, where it is called linkage learning (see (Pelikan et al., 2000), (Heckendorn and Wright, 2004)).

The correct structure for a function may be discovered from the training data using an iterative approach of adding and removing weights as training progresses. The basics of the structure discovery algorithm are to train a partial network, test the significance of the weights it contains, remove those that are not significant, then add new weights according to some criteria. The weight picking criteria chosen for this work are based on maintaining a probability distribution over the possible weights, which is updated on each round of learning so that connection orders and neurons that have proved useful in previous rounds have a higher probability of being picked in subsequent rounds. The process is described in algorithm 4.

Algorithm 4: Probability distribution based structure discovery algorithm.

```
Start with an empty network with weight set $W = \emptyset$
Initialise a distribution over possible weights, $P(w)$
repeat
    Sample new weights from the distribution $P(w)$
    Calculate the values for all weights
    Remove any insignificant weights from the network
    Update the weights distribution, $P(w)$
    Calculate the test error
until The test error is sufficiently low or doesn't change
```

The probability distribution over W is based on the order of a weight and the neurons it connects. The sampling process involves first picking an order, k for the weight to be added from a distribution, $Pk()$ over all possible orders $(1 \ldots n)$ and then picking k neurons to connect from a distribution, $Pn()$ over the n available neurons. The choice made for the algorithm described here is to impose an exponential distribution on the choice of weight order, centred at order c where initially $c = 1$ and c is incremented as lower order weights are either used or discarded.

$$Pk(k) = \lambda e^{-\lambda|c-k|} \tag{21}$$

where λ controls the width of the distribution. 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, $Pk(k)$ is updated in two ways. Firstly, c is incremented 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.

The weight order probability distribution, $Pk(k)$ is updated by counting the proportion of weights in the current network that are of each order. Let this vector of proportions be $\mathbf{p} = p_1 \ldots p_n$ where p_i is the number of weights at order i divided by the total number of weights in the network. These proportions are then used to update $Pk()$ as follows along with an updated version of the exponential distribution:

$$Pk(i) \leftarrow (1-(\alpha+\beta))Pk(i) + \alpha p_i + \beta\lambda e^{-\lambda|c-k|} \tag{22}$$

where α and β are update rates such that $0 \leq \alpha \leq 1$, $0 \leq \beta \leq 1$ and $0 < \alpha+\beta \leq 1$.

The neuron distribution update rule sets the probability of a neuron being picked to be proportionate to the sum of the absolute values of the weights connected to it. The contribution of neuron i is $C(i)$

$$C(i) = \sum_{x_i \in w} |w| \tag{23}$$

and the probability of picking neuron i is

$$Pn(i) = \frac{C(i)}{\sum_{j=0}^{n-1} C(j)} \tag{24}$$

As each new set of weights is added, another phase of learning cycles is required to update the new network. The existing weight values will be close to their correct value, but need to change slightly to accommodate the newly added weights. The new weights also need to be learned. This is requires an on line learning approach as the existing weights need to be moved from their current values, rather than calculated from scratch, making the delta rule the ideal choice. Weights may be identified for removal by performing a t-test on their values, looking for significant difference from zero in order to keep a weight. Alternatively, each new structure may be learned from scratch using LASSO, which has the advantage of automatically identifying the weights to remove as those with a coefficient of zero.

This approach to growing a neural network differs from the many previously reported methods in that it only adds weights, not neurons. Generally, grow-and-learn neural network algorithms proceed by adding neurons to the existing hidden layer or by introducing new layers. For example the Netlines algorithm (Torres-Moreno and Gordon, 1998) adds binary hidden units one at a time in an incremental approach to learning classifier functions. The Upstart algorithm (Frean, 1990), on the other hand, produces deeper tree structured networks by adding pairs of hidden units between the input layer and the current first hidden layer. As a MOHN contains no hidden units, it is only weights, not neurons that are added at each iteration of the growth algorithm.

The MOHN may also be compared to other statistical models. By introducing a link function, a MOHN becomes a generalised linear model (GLM) (Dobson and Barnett, 2011). A link function is usually a non-linear function that is applied to the output of a linear regression to allow a wider range of probability distributions to be modelled. The general form is

$$g(\mathbf{x}) = \hat{y} \tag{25}$$

where $\hat{y}$ is the energy function of the network from equation 9 and $g(\mathbf{x})$ is the link function. If the inverse of $g(\mathbf{x})$ is known, then the learning rules may all be used with the simple replacement of $f(\mathbf{x})$ with $g^{-1}(f(\mathbf{x}))$. For example, setting $g(\mathbf{x}) = e^{\mathbf{x}}$ and $g^{-1}(\mathbf{x}) = \ln(\mathbf{x})$ constructs a Boltzmann distribution if the target values, $f(\mathbf{x})$ are proportional to the probability of pattern $\mathbf{x}$. This is the approach taken in (Shakya et al., 2012) who use a Markov Random Field to model the distribution of solutions to optimisation problems by training a mixed order network with an exponential link function using OLS.

5 EXPERIMENTAL RESULTS

In this section, the learning rules described in this paper are compared with each other and with a standard multi layer perceptron (MLP) for the speed at which they learn. The Hamming distance based function of equation 15 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 thousands of different functions of varying degrees of complexity.

5.1 Speed Against Complexity

One way to vary the complexity of a function is to vary the number of turning points it contains. This section describes a set of experiments designed to measure the speed of learning of each of the MOHN learning rules and an MLP as the complexity of the function to be learned varies. Each single experiment involved training a MOHN and an MLP on a data set generated from a function with a random number of turning points. The same data was used to train three different MOHNs, one with each learning rule from OLS, LASSO and LDR. The function had 15 inputs and the MOHNs were fully connected up to order three, giving them 576 weights. The MLP has only 10 hidden units, giving it only 176 weights.

A sample of 580 random points was used for training each network. For the iterative learning methods (all except OLS) a target error of 0.01 was used as a stopping criteria, hence the measure of interest was time taken to reach a training error of 0.01. This process was repeated 1000 times, each with a new function with a number of turning points between 1 and 30.

Figure 3 shows the results. All methods except the MLP learned the function in a constant time, regardless of the degree of complexity. The MLP was able to learn the single turning point function (i.e. linear function) in less time than it was able to learn the more complex functions. The function with two turning points was also faster than those with more. After two turning points, the learning time for the MLP became constant. Regardless of the complexity of the function, the MLP always took considerably longer, followed by OLS. The LDR and LASSO algorithms took similar amounts of time and were the fastest.

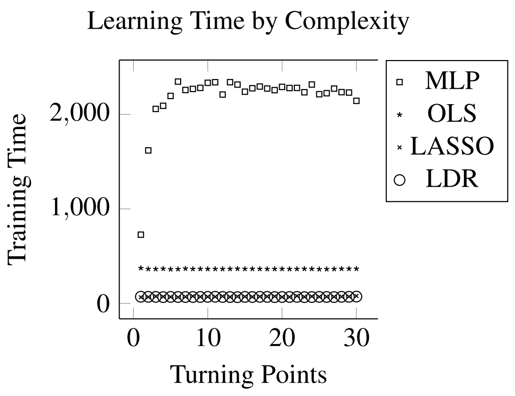

Figure 3: Average learning time in milliseconds by function complexity for different MOHN learning rules and an MLP. The LASSO and LDR values are almost equal and can be seen along the bottom of the graph.

5.2 Speed By Network Size

Another set of similar experiments related the training speed of each method to the size of the network. The number of inputs to a network was varied from 5 to 15 and 1000 trials were run. The mean squared error of the result of performing OLS was used as the stopping criteria for the MLP and the MOHN as it was trained with LDR, ensuring that all models had the same level of accuracy. Figure 4 shows the results. OLS is known to have a time complexity of $O(np^2)$ where n is the number of data points and p is the number of variables. LASSO and LDR were of the same order, but the algorithms ran in less time. The MLP's training time grew exponentially with the number of variables in these particular experiments. As before the MOHN models all reached the target training error faster than the MLP.

5.2.1 Error Descent Rate

The difference in training speed between the MOHN and an MLP was investigated further by recording the average error by training epoch for the first twenty passes through the training data. Figure 5 shows the average error on each pass of the training data from 1000 repeated trials on functions of varying complexity. The error bars show 1 standard deviation from the mean. Note that the MOHN error drops faster and that there is far less variation across trials (the error

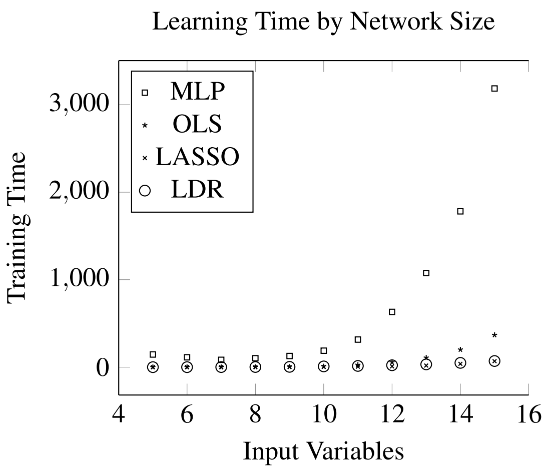

Figure 4: Average learning time in milliseconds by number of inputs for different MOHN learning rules and an MLP.

bars for the MOHN are sufficiently short that they sit inside the marks).

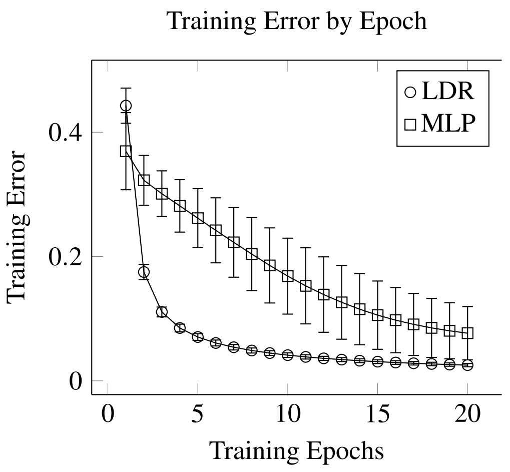

Figure 5: The mean and standard deviation of training error as it descends over twenty training epochs, comparing an MLP with the Linear Delta Rule training a MOHN.

The improved learning speed of the MOHN and the slower, more varied speed of the MLP may be explained by the fact that the MLP combines fitting parameter values with feature selection. Recently, (Swingler, 2014) provided an insight into the phases of MLP training, showing that early training cycles are taken up with fixing the role of the hidden units and later cycles then fit the parameters within the constraints of the features encoded by those hidden units. The MOHN does not have hidden units and so only needs to fit parameter values to its fixed structure. Of course, that structure needs to be discovered, but the task of structure discovery and parameter fitting are separated, unlike the case for the MLP.

Another consequence of the MLP's dual learning

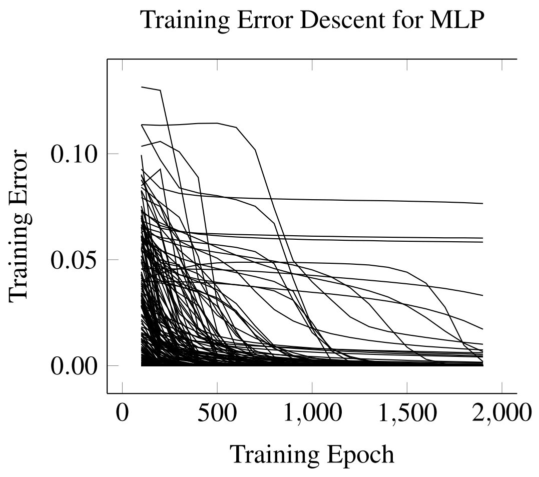

Figure 6: Traces of training error over 200 different attempts at training an MLP on a concatenated XOR function. Note the variation in convergence time and the presence of a number of failed attempts after 2000 epochs.

task of fitting both function structure and regression fit is that the error function contains local minima. These occur when the hidden units encode a suboptimal set of features and the network fits weight values to them. This is commonly solved by re-starting the training process from a different random set of initial weight values. The MOHN error function does not contain local minima, so the weights do not need to be randomised before learning, as shown in algorithm 3. To illustrate this point, a final set of experiments compared an MLP trained with error back propagation to a MOHN trained with the LDR on a function designed to contain local minima in its cost function. The function to be learned was a concatenation of XOR pairs such 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, so 101010 would produce an output of one and 110011 would produce zero. Figure 6 shows the traces of 200 MLPs started with random weights, each trained for 2000 cycles through the training data. The variation in error descent is clear, with some networks converging quickly, some taking many training epochs to converge, and some still stuck in local minima after 2000 epochs.

For functions of small numbers of inputs, it was possible to exhaustively sample the function space and so use the weighted Hebb rule to allow the MOHN to learn the function fully in a single pass of the data. For networks where the number of inputs is too large for an exhaustive sample, a random sample was taken. Figure 7 shows the trace of the training error during 200 attempts at learning the same XOR based function as that in figure 6 using a MOHN with the LDR. The variation is not due to random starting points—all networks start with weights at zero—but is due to the fact that the training data is a small random subset of the full input space. Note that there are no traces that indicate a local minimum; all go to zero error.

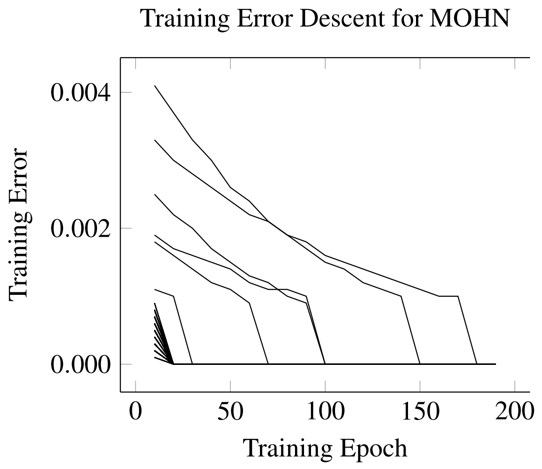

Figure 7: Traces of training error over 200 different attempts at training a MOHN on a concatenated XOR function. Compare both the scale of the error and the number of training epochs involved with the same plot for the MLP in figure 6.

6 SUMMARY AND FUTURE DIRECTIONS

Mixed Order Hyper Networks are universal function approximators over $f : \{-1,1\}^n \to \mathbb{R}$. They may be trained from a sample of data to act as either a regression function that attempts to fit the function that underlies the data across the entire function space or just to capture the function's turning points as energy minima. Learning may be off line, in which case all of the data needs to be available at one time, or on line in situations where data is streamed or the network structure is changing and existing weights need to be updated. This paper presented five learning rules designed to cover both on line and off line learning, and both regression and content addressable memory learning. Other learning methods might also be considered such as ridge regression or LARS, but that is left for future work.

This paper has only presented networks for function learning, but they may also be used as classifiers. As a MOHN has only one output, binary classifications are straight forward. Further work is required to discover the best way to learn multi-class models. Learning a classifier will also introduce the possibility of using alternative learning algorithms such as Minimerror (Torres-Moreno et al., 2002), which is a perceptron learning rule with a cost function designed to reduce the number of classification errors rather than mean squared error. The MOHN and its learning rules would also be usefully compared to deep networks as they present a start contrast in approach.

The issue of MOHN structure discovery was also raised, but the detail is left for future work. The experiments presented in this paper worked on the assumption that the networks in question contained weights of sufficient order to capture the functions on which they were trained. This becomes increasingly difficult as the number of inputs grows. Problems with large numbers of inputs require a structure discovery phase to be carried out as part of the training process.

With a given network structure, training a MOHN is faster and has less error variance across trials than training with an MLP. Additionally, the training algorithm has no local minima when training a fixed structure MOHN, making training more reliable than that of an MLP. Of course, any algorithm used to discover the correct structure for the MOHN may well have local optima, but that (again) is a matter for future work.

REFERENCES

Beauchamp, K. (1984). *Applications of Walsh and Related Functions*. Academic Press, London.

Caparrós, G. J., Ruiz, M. A. A., and Hernández, F. S. (2002). Hopfield neural networks for optimization: study of the different dynamics. *Neurocomputing*, 43(1-4):219–237.

Dobson, A. J. and Barnett, A. (2011). *An introduction to generalized linear models*. CRC press.

Frean, M. (1990). The upstart algorithm: A method for constructing and training feedforward neural networks. *Neural computation*, 2(2):198–209.

Hastie, T., Tibshirani, R., Friedman, J., Hastie, T., Friedman, J., and Tibshirani, R. (2009). *The elements of statistical learning*, volume 2. Springer.

Heckendorn, R. B. and Wright, A. H. (2004). Efficient linkage discovery by limited probing. *Evolutionary computation*, 12(4):517–545.

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences USA*, 79(8):2554–2558.

Hopfield, J. J. and Tank, D. W. (1985). Neural computation of decisions in optimization problems. *Biological Cybernetics*, 52:141–152.

Kubota, T. (2007). 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.

Pelikan, M., Goldberg, D. E., and Cantú-paz, E. E. (2000). Linkage problem, distribution estimation, and bayesian networks. *Evolutionary Computation*, 8(3):311–340.

Shakya, S., McCall, J., Brownlee, A., and Owusu, G. (2012). Deum - distribution estimation using markov networks. In Shakya, S. and Santana, R., editors, *Markov Networks in Evolutionary Computation*, volume 14 of *Adaptation, Learning, and Optimization*, pages 55–71. Springer Berlin Heidelberg.

Swingler, K. (2012). On the capacity of Hopfield neural networks as EDAs for solving combinatorial optimisation problems. In *Proc. IJCCI (ECTA)*, pages 152–157. SciTePress.

Swingler, K. (2014). A walsh analysis of multilayer perceptron function. In *Proc. IJCCI (NCTA)*, pages –.

Swingler, K. and Smith, L. (2014a). Training and making calculations with mixed order hyper-networks. *Neurocomputing*, (141):65–75.

Swingler, K. and Smith, L. S. (2014b). An analysis of the local optima storage capacity of hopfield network based fitness function models. *Transactions on Computational Collective Intelligence XVII, LNCS 8790*, pages 248–271.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288.

Torres-Moreno, J.-M., Aguilar, J., and Gordon, M. (2002). Finding the number minimum of errors in n-dimensional parity problem with a linear perceptron. *Neural Processing Letters*, 1:201–210.

Torres-Moreno, J.-M. and Gordon, M. B. (1998). Efficient adaptive learning for classification tasks with binary units. *Neural Computation*, 10(4):1007–1030.

Venkatesh, S. S. and Baldi, P. (1991). Programmed interactions in higher-order neural networks: Maximal capacity. *Journal of Complexity*, 7(3):316–337.

Walsh, J. (1923). A closed set of normal orthogonal functions. *Amer. J. Math*, 45:5–24.

Wilson, G. V. and Pawley, G. S. (1988). On the stability of the travelling salesman problem algorithm of hopfield and tank. *Biol. Cybern.*, 58(1):63–70.