Neural Network Training with Constrained Integer Weights

V.P. Plagianakos
University of Patras
Department of Mathematics,
U.P. Artificial Intelligence
Research Center (UPAIRC),
GR-26500 Patras, Greece.
e-mail: vpp@math.upatras.gr

M.N. Vrahatis
University of Patras
Department of Mathematics,
U.P. Artificial Intelligence
Research Center (UPAIRC),
GR-26500 Patras, Greece.
e-mail: vrahatis@math.upatras.gr

Abstract- In this contribution we present neural network training algorithms, which are based on the differential evolution (DE) strategies introduced by Storn and Price [Journal of Global Optimization 11, 341–359, 1997]. These strategies are applied to train neural networks with small integer weights. Such neural networks are better suited for hardware implementation than the real weight ones. Furthermore, we constrain the weights and biases in the range $[-2^k + 1, 2^k - 1]$, for $k = 3, 4, 5$. Thus, they can be represented by just k bits.

These algorithms have been designed keeping in mind that the resulting integer weights require less bits to be stored and the digital arithmetic operations between them are easier to be implemented in hardware. Obviously, if the network is trained in a constrained weight space, smaller weights are found and less memory is required. On the other hand, as we have found here, the network training procedure can be more effective and efficient when large weights are allowed. Thus, for a given application a trade off between effectiveness and memory consumption has to be considered.

Our intention is to present results of evolution algorithms on this difficult task. Based on the application of the proposed class of methods on classical neural network benchmarks, our experience is that these methods are effective and reliable.

1 Introduction

Artificial Feedforward Neural Networks (FNNs) have been widely used in many application areas in recent years and have shown their strength in solving hard problems in Artificial Intelligence. Although many different models of neural networks have been proposed, multilayered FNNs are the most common. FNNs consist of many interconnected identical simple processing units, called neurons. Each neuron calculates the dot product of the incoming signals with its weights, adds the bias to

the resultant, and passes the calculated sum through its activation function. In a multilayer feedforward network the neurons are organized into layers with no feedback connections.

FNNs can be simulated in software, but in order to be utilized in real life applications, where high speed of execution is required, hardware implementation is needed. The natural implementation of an FNN – because of its modularity – is a parallel one. The problem is that the conventional multilayer FNNs, which have continuous weights, are expensive to implement in digital hardware. Another major implementation obstacle is the weight storage. FNNs having integer weights and biases are easier and less expensive to implement in electronics as well as in optics and the storage of the integer weights is much easier to be achieved.

Another advantage of the FNNs with integer weights is their immunity to noise in the training data. Such networks only capture the main feature of the training data. Low amplitude noise that possibly contaminates the training data cannot perturb the discrete weights, because those weights require relatively large variations to jump from one integer value to another.

In recent publications [6, 5] we have studied neural networks with integer weights. Here, we proceed further by studying neural networks having integer weights constrained in the ranges $[-2^k + 1, 2^k - 1]$, $k = 3, 4, 5$ which correspond to k-bit integer representation of the weights. This property reduces the amount of memory required for weight storage in digital electronic implementations. Additionally, it simplifies the digital multiplication operation, since multiplying any number with a k-bit integer requires only the following number of basic instructions: one sign change, $(k-1)(k-2)/2$ one–step left shifts and $k - 2$ additions. Finally, if inputs are restricted to the set $\{-1, 1\}$ (bipolar inputs), the neurons in the first hidden layer require only sign changes during multiplication operations, and only integer additions.

The efficient supervised training of FNNs, i.e. the incremental adaptation of the connection weights that propagate information between the neurons, is a subject of considerable ongoing research and numerous algorithms have been proposed to this end. The majority

of those algorithms use the negative of the gradient of the error function, $-\nabla E(w)$, as their descent direction. The gradient $\nabla E(w)$ can be computed by the BackPropagation of the error through the layers of the network. This calculation, however, is computationally expensive and difficult to be implemented in hardware. In this contribution, we propose a new class of training algorithms that do *not* need the gradient of E.

Formally, a typical FNN consists of L layers, where the first layer denotes the input, the last one, L, is the output, and the intermediate layers are the hidden layers. It is assumed that the (l-1) layer has N_{l-1} neurons. The neurons operate according to the following equations

$$net_j^l = \sum_{i=1}^{N_{l-1}} w_{ij}^{l-1,l} y_i^{l-1} + \theta_j^l, \quad y_j^l = \sigma^l\left(net_j^l\right),$$

where $w_{ij}^{l-1,l}$ is the connection weight from the i-th neuron at the ($l-1$) layer to the j-th neuron at the l-th layer, y_i^l is the output of the ith neuron belonging to the l-th layer, θ_j^l denotes the bias of the j-th neuron at the lth layer, and σ is a nonlinear activation function. The weights in the FNN can be expressed in vector notation. Let the weight vector have the form: $w = (w_1, w_2, \ldots, w_N)$. The weight vector, in general, defines a point in the N–dimensional real Euclidean space $\mathbb{R}^N$, where N denotes the total number of weights and biases in the network. Throughout this paper w is considered to be the k–*bit*, $k = 3, 4, 5$ *integer* vector of the weights and biases.

From the optimization point of view, supervised training of an FNN is equivalent to minimizing a global error function, which is a multivariate function that depends on the weights in the network. The square error over the set of input–desired output patterns with respect to every weight, is usually taken as the function to be minimized. Specifically, the error function for an input pattern t is defined as follows:

$$e_j(t) = y_j^L(t) - d_j(t), \quad j = 1, 2, \ldots, N_L,$$

where $d_j(t)$ is the desired response of an output neuron at the input pattern t. For a fixed, finite set of input-desired output patterns, the square error over the training set which contains T representative pairs is:

$$E(w) = \sum_{t=1}^{T} E_t(w) = \sum_{t=1}^{T} \sum_{j=1}^{N_L} e_j^2(t),$$

where $E_t(w)$ is the sum of the squares of errors associated with the pattern t. Minimization of E is attempted by using a training algorithm to update the weights. Efficient training algorithms have been proposed for trial and error based training, but it is difficult to use them when training with discrete weights [1, 2].

In this work a differential evolution approach, as explained in Section 2, has been utilized to train neural networks with k–bit, $k = 3, 4, 5$ integer weights, suitable for hardware implementation. A brief overview of the most used differential evolution strategies is also presented. Experiments and computer simulation results are presented in Section 3. In Section 3, in addition to the speed and robustness, we also evaluate the generalization capabilities of 3–bit integer weight neural networks, by testing them on the MONK's problems [8]. The final section contains concluding remarks and a short discussion for future work.

2 Training neural networks with integer weights

In a recent work, Storn and Price [7] have presented a novel minimization method, called Differential Evolution (DE), which has been designed to handle non-differentiable, nonlinear and multimodal objective functions. To fulfill this requirement, DE has been designed as a stochastic parallel direct search method, which utilizes concepts borrowed from the broad class of evolutionary algorithms, but requires few easily chosen control parameters. Experimental results have shown that DE has good convergence properties and outperforms other evolutionary algorithms.

In order to apply DE to neural network training with k–bit integer weights, we start with a specific number (NP) of N-dimensional integer weight vectors, as an initial weight population, and evolve them over time. NP is fixed throughout the training process. The weight population is initialized with random integers from the interval $[-2^k + 1, 2^k - 1]$, for $k = 3, 4, 5$ following a uniform probability distribution.

At each iteration, called *generation*, new weight vectors are generated by the combination of weight vectors randomly chosen from the population and the outcome is rounded to the nearest integer. Moreover, we force the new vectors to be in the range $[-2^k + 1, 2^k - 1]^N$. This operation is called *mutation*. The outcoming k–bit integer weight vectors are then mixed with another predetermined integer weight vector – the *target* weight vector – and this operation is called *crossover*. This operation yields the so-called *trial* weight vector, which is an integer vector in the range $[-2^k + 1, 2^k - 1]^N$. The trial vector is accepted for the next generation if and only if it reduces the value of the error function E. This last operation is called *selection*. We now briefly review the two basic DE operators used for integer weight FNN training.

2.1 The mutation operator

The first DE operator we consider is mutation. Specifically, for each weight vector w_g^i, $i = 1, \ldots, NP$, where g

denotes the current generation, a new vector v^i_{g+1} (mutant vector) is generated according to one of the following relations:

$$v^i_{g+1} = w^{r_1}_g + \mu \left(w^{r_1}_g - w^{r_2}_g \right), \tag{1}$$

$$v^i_{g+1} = w^{best}_g + \mu \left(w^{r_1}_g - w^{r_2}_g \right), \tag{2}$$

$$v^i_{g+1} = w^{r_1}_g + \mu \left(w^{r_2}_g - w^{r_3}_g \right), \tag{3}$$

$$v^i_{g+1} = w^i_g + \mu \left(w^{best}_g - w^i_g \right) + \mu \left(w^{r_1}_g - w^{r_2}_g \right), \tag{4}$$

$$v^i_{g+1} = w^{best}_g + \mu \left(w^{r_1}_g - w^{r_2}_g \right) + \mu \left(w^{r_3}_g - w^{r_4}_g \right), \tag{5}$$

$$v^i_{g+1} = w^{r_1}_g + \mu \left(w^{r_2}_g - w^{r_3}_g \right) + \mu \left(w^{r_4}_g - w^{r_5}_g \right), \tag{6}$$

where w^{best}_g is the best member of the previous generation, $\mu > 0$ is a real parameter, called mutation constant, which controls the amplification of the difference between two weight vectors and

$$r_1, r_2, r_3, r_4, r_5 \in \{1, 2, \ldots, i-1, i+1, \ldots, NP\},$$

are random integers mutually different and different from the running index i. Obviously, the mutation operator results in a real weight vector. As our aim is to maintain an integer weight population at each generation, each component of the mutant weight vector is rounded to the nearest integer. Additionally, if the mutant vector is not in the range $[-2^k + 1, 2^k - 1]^N$, we take:

$$v^i_{g+1} = \text{sign}(v^i_{g+1}) \times \left(\left| v^i_{g+1} \right| \mod 2^{k-1} \right).$$

Relation (1) has been introduced as crossover operator for genetic algorithms [4] and is similar to relations (2) and (3). The remaining relations are modifications which can be obtained by the combination of (1), (2) and (3). It is clear that more such relations can be generated using the above ones as building blocks. In a previous work [6], we have shown that the above relations can efficiently be used to train FNNs with arbitrary integer weights.

2.2 The crossover operator

To increase further the diversity of the rounded mutant weight vector, the crossover operator is applied. Specifically, for each integer component j ($j = 1, 2, \ldots, N$) of the mutant weight vector v^i_{g+1}, we randomly choose a real number r from the interval $[0, 1]$. Then, we compare this number with ρ (crossover constant), and if $r \leq \rho$ we select, as the j-th component of the trial vector u^i_{g+1}, the corresponding component j of the mutant vector v^i_{g+1}. Otherwise, we pick the j-th component of the integer target vector w^i_{g+1}. It must be noted that the result of this operation is again a k-bit integer vector.

3 Functionality tests

Two classical learning test problems – the eXclusive OR (XOR) and the 3-Bit Parity problems – have been used for testing the functionality, and computer simulations have been developed to study the performance of the DE training algorithms for various values of k. In order to examine the generalization behavior of these algorithms, we have tested the best of them on the well known MONK's problems [8].

The simulations have been carried out on a IBM PC compatible, using MATLAB version 5.2. For each of the test problems we present a table summarizing the performance of the DE algorithms using different mutation rules. We call DE1 the algorithm that uses relation (1) as mutation operator, DE2 the algorithm that uses relation (2), and so on.

The reported parameters for simulations that have reached solution are: *min* the minimum number of error function evaluations, *mean* the mean value of error function evaluations, *max* the maximum number of error function evaluations, *s.d.* the standard deviation of error function evaluations, and *succ.* simulations succeeded out of 1000 within the generation limit *maxgen*.

When an algorithm fails to converge within the *maxgen* limit, it is considered that it fails to train the FNN and its error function evaluations are not included in the statistical analysis of the algorithms. We must note here that a key feature of the DE algorithms is that *only* error function values are needed. No gradient information is required, so there is no need of backward passes.

For all the simulations we have used bipolar input and output vectors and hyperbolic tangent activation functions in both the hidden and output layer neurons. We made no effort to tune the mutation and crossover parameters, μ and ρ respectively. Fixed values ($\mu = 0.5$ and $\rho = 0.7$) have been used instead. The weight population has been initialized with random integers from the interval $[-2^k + 1, 2^k - 1]$.

The weight population size NP has been chosen to be twice the dimension of the problem, i.e. $NP = 2N$, for all the simulations. Some experimental results have shown that a good choice for NP is $2N \leq NP \leq 4N$. It is obvious that the exploitation of the weight space is more effective for large values of NP, but sometimes more error function evaluations are required. On the other hand, small values of NP make the algorithm inefficient and more generations are required in order to converge to the minimum.

For the test problems considered, no choice of the parameters has been needed in order to obtain optimal or at least nearly optimal convergence speed. The parameters we have used have fixed values and we have made no effort to tune them. Of course, one can try to tune the μ, ρ and NP parameters to achieve better results, i.e. less error function evaluations and/or exhibit higher success rates.

3.1 The eXclusive-OR problem

The first test problem we will consider is the eXclusive-OR (XOR) Boolean function problem, which historically has been considered as a good test of a network model and learning algorithm. The XOR function maps two binary inputs to a single binary output. This simple Boolean function is not linearly separable and thus requires the use of extra hidden units to learn the task. Moreover, it is sensitive to initial weights as well as to learning rate variations and presents a multitude of local minima with certain weight vectors. A 2-2-1 FNN (six weights, three biases) has been used for these simulations and the training has been stopped when the value of the error function E, has been $E \leq 0.1$ within $maxgen$=100 generations. The population size is NP=18.

Table 1: Results for the XOR problem

k	Alg.	min	mean	max	s.d.	succ.
3	DE1	54	191.9	810	89.7	63.0%
	DE2	90	836.3	1782	371.7	93.5%
	DE3	90	300.5	1584	171.2	82.5%
	DE4	54	364.5	1676	222.6	93.4%
	DE5	36	1047.8	1782	422.6	63.0%
	DE6	54	931.5	1782	431.1	73.1%
4	DE1	90	182.9	612	97.5	69.0%
	DE2	72	266.7	1188	162.4	87.2%
	DE3	198	647.8	1566	246.3	99.1%
	DE4	126	764.1	1656	357.2	95.9%
	DE5	72	316.9	1602	300.4	92.3%
	DE6	108	660.0	1566	368.7	93.1%
5	DE1	54	192.9	684	124.7	75.1%
	DE2	72	284.9	1332	216.2	80.4%
	DE3	144	583.9	1314	256.3	97.3%
	DE4	180	706.1	1764	343.7	97.5%
	DE5	72	300.5	1584	250.2	85.2%
	DE6	90	482.9	1368	264.9	93.0%
NP=18, $\mu = 0.5$, $\rho = 0.7$, $maxgen$=100						

The results of the simulation are shown in Table 1. A typical 3–bit weight vector after the end of the training process is $w = (3, 3, 2, 3, 2, -2, 1, -3, -2)$ and the corresponding value of the error function is $E = 0.0221$. The six first components of the above vector are the weights and the remaining three are the biases.

It must be noted that, the success rates of all strategies are better than other well-known continuous weight training algorithm, such as BackPropagation (BP), adaptive BP or BP with momentum.

3.2 3-Bit parity

The second test problem is the parity problem, which can be considered as a generalized XOR problem but is more difficult. The task is to train a neural network to produce the sum, mod 2, of 3 binary inputs – otherwise known as computing the "odd parity" function. We use a 3-3-1 FNN (twelve weights, four biases) in order to train the 3-Bit Parity problem. The initial population consists of 32 weight vectors. The results of the computer simulation are summarized in the Table 2.

A typical 3–bit weight vector after the end of the training process is $w = (3, 3, 2, 3, -1, -1, 2, -2, -2, -3, 3, -3, 1, 0, 1, 1)$ and the corresponding value of the error function is $E = 0.0257$.

Table 2: Results for the 3-Bit parity problem

k	Alg.	min	mean	max	s.d.	succ.
3	DE1	96	809.2	2016	313.9	88.1%
	DE2	704	1966.9	3072	769.2	1.5%
	DE3	320	1123.4	3168	461.0	97.7%
	DE4	160	2072.1	3168	631.9	75.5%
	DE5	1344	2057.1	3072	703.9	0.7%
	DE6	160	1890.0	3168	907.7	3.2%
4	DE1	96	762.7	2624	515.1	90.3%
	DE2	192	2056.0	3072	711.4	28.1%
	DE3	320	978.3	3136	555.5	95.4%
	DE4	288	1333.0	3104	652.6	96.5%
	DE5	192	1959.1	2816	981.6	9.6%
	DE6	1056	2153.4	3168	619.7	17.8%
5	DE1	160	622.6	3072	522.1	90.8%
	DE2	576	1994.1	3168	657.6	60.8%
	DE3	224	896.3	2688	450.6	99.1%
	DE4	256	1060.2	3168	716.6	97.5%
	DE5	672	2112.0	3104	644.9	26.0%
	DE6	352	2062.5	3168	794.8	44.1%
NP=32, $\mu = 0.5$, $\rho = 0.7$, $maxgen$=100						

In a previous work [6], we have shown that DE3 and DE4 are the more suitable algorithms for integer weight training. This is also evident form the results shown in Table 2. In this problem the DE1, DE3 and DE4 algorithms exhibit excellent over all performance for all the values of k tested.

3.3 Generalization performance

In addition to training speed and efficiency, we have also evaluated the generalization performance of the DE algorithms.

To this end, we have tested the best of them (DE3 and DE4) in their most constrained form ($k = 3, w \in [-3, 3]^N$) on the MONK's problems. These are difficult binary classification tasks which have been used for comparing the generalization performance of learning algorithms. These problems rely on the artificial robot domain, in which robots are described by six different attributes. Each problem is given by a logical description of the class, as shown below:

MONK-1: (*Attribute1* = *Attribute2*) OR (*Attribute5* = 1). This problem is in standard Disjunctive Normal Form (DNF). 124 examples have been selected randomly from the data set for training, while the remaining 308 have been used for the generalization testing. There are no misclassifications.

MONK-2: (Only two attributes = 1). This problem is similar to the parity problem mentioned above and is difficult to describe in DNF or Conjunctive Normal Form (CNF). 169 examples have been randomly selected from the data set for training, while the rest have been used for testing. Again, there is no noise.

MONK-3: (*Attribute5* = 3 AND *Attribute4* = 1) OR (*Attribute5* $\neq$ 4 AND *Attribute2* $\neq$ 3) with added noise. This problem is also in DNF but with 5% deliberate misclassifications in the training set, which consists of 122 examples. The remaining 310 examples have been used for testing.

Each one of the six attributes can have one of 3, 3, 2, 3, 4, and 2 values, respectively, which results 432 possible combinations that constitute the total data set (see [8], for details). Finally, each possible value for every attribute is assigned a single bipolar input, resulting 17 inputs.

We have tested DE3 and DE4 against the BackPropagation (BP), the BackPropagation with Weight Decay (BPWD), and the Cascade Correlation (CC) algorithms. In Table 3 we exhibit the comparative results on the MONK's problems.

Table 3: Comparison of generalization performance on the MONK's problems

Algorithm	MONK-1	MONK-2	MONK-3
BP	100%	100%	93.1%
BPWD	100%	100%	97.2%
CC	100%	100%	97.2%
DE3	100%	100%	100%
DE4	100%	100%	100%

It is clear from Table 3 that the DE algorithms generate FNNs, which are at least as capable as the best generated by real-weight learning algorithms. Those networks, in all the MONK's problems, seem to have learned the concept embedded in the training data. This is more evident in MONK-3, where there are 5% deliberate misclassifications and the networks generated by BP, BPWD, and CC seem to fail to capture the concept embedded in the training data, and fit to the noise instead.

The topology of the trained networks is shown in Table 4. It is known that the best generalizers are neither too complex nor too simple; they exactly match the complexity of the embedded in the training data concept.

We think that the reason why our algorithms, in general, need a bigger network in order to generate FNNs with good generalization capabilities is that more integers than real numbers are needed in order to match the complexity of the given problem.

Table 4: Network configuration for the MONK's problems

Algorithm	MONK-1	MONK-2	MONK-3
BP	17:3:1	17:2:1	17:4:1
BPWD	17:2:1	17:2:1	17:2:1
CC	17:1:1	17:1:1	17:3:1
DE3	17:4:1	17:4:1	17:3:1
DE4	17:4:1	17:4:1	17:3:1

At the end of this paper, we present three tables with the integer weights of the networks trained using DE3. Similar networks are generated by DE4.

4 Concluding remarks and discussion

Differential evolution based algorithms for k-bit, $k = 3, 4, 5$ integer weight neural network training are studied in this contribution. This is an interesting kind of neural networks, because the amount of memory required for the storage of their weights is significantly reduced compared to networks with real weights. Additionally, the digital multiplication operations required are simplified.

Furthermore, it is known that the hardware implementation of the backward passes, which compute the gradient of the error function, is more difficult than the implementation of the forward passes. That is why all the proposed algorithms require only forward passes resulting in the value of the error function.

Obviously, the smaller the k the less the memory required. On the other hand, as we have observed, the network training procedure can be more effective and efficient when more bits are used. Thus, for a given application a trade off between effectiveness and memory consumption has to be considered.

In this paper, customized differential evolution operators have been applied on the population of k-bit integer weight vectors, in order to evolve them over time and exploit the constrained weight space as wide as possible. Also, the performance of these algorithms has been examined and simulation results from some classical test problems have been presented. Summarizing the simulations, we have concluded that the DE3 and DE4 algorithms are definitely the best choices for all the constrained weight spaces tested. On the other hand, even the algorithm DE1, based on the simple strategy (1), performed very well. An interesting observation is that all the algorithms considered increase their performance when the k value is increased.

The results indicate that this new class of algorithms

is promising and effective, even when compared with other well-known algorithms that require the gradient of the error function and train the network with real weights. Moreover, we have tested the generalization capabilities of the networks generated by DE3 and DE4. Both algorithms have exhibited excellent performance and have outperformed other well known real weight learning algorithms.

Preliminary investigations suggest that the methods of this paper can also be applied to discrete neural networks [3]. This kind of networks are based on neurons whose output can be in a particular state and are important, since they can handle many of the inherently binary tasks that neural networks are used for. Their internal representations are clearly interpretable, they are computationally simpler to understand than networks with sigmoid units and provide a starting point for the study of the neural network properties. In a future work we intend to present results in this approach.

Bibliography

[1] A.H. Khan, *Feedforward Neural Networks with Constrained Weights*, Ph.D. Thesis, Univ. of Warwick, Dept. of Engineering, (1996).

[2] A.H. Khan and E.L. Hines, Integer-weight neural nets, *Electronics Letters*, **30**, 1237–1238, (1994).

[3] G.D. Magoulas, M.N. Vrahatis, T.N. Grapsa and G.S. Androulakis, A training method for discrete multilayer neural networks, In: *Mathematics of Neural Networks, Models, Algorithms and Applications*, S.W. Ellacott, J.C. Mason and I.J. Anderson Eds., Kluwer Academic Publishers, 250–254, (1997).

[4] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, (1996).

[5] V.P. Plagianakos and M.N. Vrahatis, Training Neural Networks with 3–bit Integer Weights, Proceedings of Genetic and Evolutionary Computation Conference, to appear, (1999).

[6] V.P. Plagianakos, D.G. Sotiropoulos and M.N. Vrahatis, Integer Weight Training by Differential Evolution Algorithms, In: *Recent Advances in circuits and systems*, N.E. Mastorakis Ed., World Scientific, (1998).

[7] R. Storn and K. Price, Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous spaces, *Journal of Global Optimization*, **11**, 341–359, (1997).

[8] S.B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Dzeroski, S.E. Fahlman, D. Fisher, R. Hamann, K. Kaufmann, S. Keller, I. Kononenko, J. Kreuziger, R.S. Michalski, T. Mitchell, P. Pachowicz, Y. Reich, H. Vafaie, W. Van de Welde, W. Wenzel, J. Wnek and J. Zhang, *The MONK's Problems: A performance comparison of different learning algorithms*, Technical Report, Carnegie Mellon University, CMU-CS-91-197, (1991).

Table 5: MONK's problem #1: weights and biases

From node	To node				
	Hid1	Hid2	Hid3	Hid4	Out
In1	3	−3	3	−1	
In2	2	−1	2	3	
In3	3	−2	−3	−2	
In4	2	−3	3	2	
In5	1	1	2	3	
In6	1	−1	−2	1	
In7	−3	3	−3	3	
In8	−2	3	−3	3	
In9	2	2	−3	3	
In10	0	2	−2	3	
In11	1	2	−3	3	
In12	1	−3	−3	3	
In13	2	2	−3	−3	
In14	0	2	−3	−3	
In15	2	2	−3	−3	
In16	−3	−3	−3	2	
In17	−1	−3	−2	2	
Bias	−2	3	3	−3	
Hid1					−3
Hid2					−1
Hid3					−3
Hid4					2
Bias					3

Table 6: MONK's problem #2: weights and biases

From node	To node				
	Hid1	Hid2	Hid3	Hid4	Out
In1	−2	−1	−3	−2	
In2	3	2	−2	3	
In3	2	2	−2	3	
In4	1	1	−3	−2	
In5	2	2	−2	3	
In6	3	2	−2	3	
In7	2	2	2	−3	
In8	1	−3	3	2	
In9	3	1	2	−2	
In10	2	2	3	3	
In11	1	1	3	3	
In12	−2	−3	2	−2	
In13	−1	3	3	3	
In14	−2	0	3	3	
In15	−1	2	3	3	
In16	2	−1	2	−3	
In17	2	2	3	2	
Bias	−1	0	3	−3	
Hid1					0
Hid2					0
Hid3					2
Hid4					−2
Bias					0

Table 7: MONK's problem #3: weights and biases

From node	To node			
	Hid1	Hid2	Hid3	Out
In1	0	0	1	
In2	−1	0	1	
In3	−1	0	−1	
In4	0	−2	0	
In5	−2	−2	2	
In6	−1	2	0	
In7	1	1	−2	
In8	−2	1	1	
In9	−2	0	0	
In10	−1	2	2	
In11	−2	2	1	
In12	1	−1	1	
In13	−2	−1	−3	
In14	1	−3	0	
In15	0	3	1	
In16	2	0	0	
In17	0	0	1	
Bias	0	1	−2	
Hid1				0
Hid2				−2
Hid3				0
Bias				2