ESANN'1995 proceedings - European Symposium on Artificial Neural Networks
Brussels (Belgium), 19-20-21 April 1995, D-Facto public., ISBN 2-9600049-3-0, pp. 241-246

Constructing Feed-Forward Neural Networks for Binary Classification Tasks

C. Campbell[†]and C. Perez Vicente[‡]
[†]Advanced Computing Research Centre, Bristol University, Bristol BS8 1TR, United Kingdom
[‡]Dept. de Física Fonamental, Universitat de Barcelona, Diagonal 647, 08028 Barcelona, Spain

Abstract. We propose an efficient procedure for constructing and training feed-forward neural networks. The procedure can be used to generate neural networks with either a single hidden layer, a cascade or a tree-structured architecture capable of performing arbitrary binary classification tasks. The procedure can also be extended to construct neural networks with binary-valued weights.

1. Introduction

In this paper we will outline a new constructive procedure for generating feed-forward neural networks with either a single hidden layer, a cascade architecture or a tree-structured network. The networks generated have an economical structure and generalise well. In addition the procedure can be readily adapted to generate networks with binary-valued weights: neural networks with binary weights are straightforward to implement in hardware and exhibit good generalisation due to the reduction in the number of free parameters in the network.

Let us consider a neural network with N input nodes labeled by index j and one output node. Suppose we wish to map inputs ξ_j^μ onto a set of targets η^μ, where μ is the pattern index and η^μ has components ± 1. Weights leading from input j to a hidden node i will be denoted W_{ij}. If we use a ± 1 updating function for the hidden nodes then an input vector S_j will give an internal representation $S_i = sign(\sum_j W_{ij} S_j - T_i)$ where T_i is the threshold at hidden node i. We will define the *sign*-function as having an output of $+1$ if its argument is greater than or equal to zero and -1 otherwise.

For binary classification tasks the patterns belong to two sets: patterns with target $\eta^\mu = 1$ (the set P^+) and those with target $\eta^\mu = -1$ (the set P^-). For binary inputs (quantised ± 1) it is always possible to find a set of weights and thresholds which will correctly store *all* the patterns belonging to one of these sets and at least one member belonging to the other set [1] (we will comment on the case of analogue input data in section 4). For example, suppose pattern $\mu = 1$ has target $+1$. If we use weights $W_{ij} = \xi_j^1$ and a threshold $T_i = N$ then $S_i = sign(\sum_j W_{ij} S_j - T_i)$ gives an output $S_i = +1$ if S_j is equal to ξ_j^1 and

-1 otherwise. Usually it is possible to do better than this and store a number of members of P^+ in addition to all the P^- . A set of weights and thresholds which correctly store all the P^+ patterns and some of the P^- will be said to induce a $\oplus$-dichotomy while a $\ominus$-dichotomy will correspond to correct storage of all the P^- patterns and some of the P^+. In section 2.3 we will describe a heuristic procedure for finding these $\oplus$ and $\ominus$ -dichotomies.

Let us consider a pair of nodes in a hidden layer with direct connections to the output node (each with weight-value $+1$). Let us assume the first of these nodes induces a $\oplus$-dichotomy and the second induces a $\ominus$-dichotomy. If the first node correctly stored a pattern belonging to P^- then the second node must similarly store this pattern correctly and both nodes contribute -1's to the output node. If the pattern belonging to P^- was not stored correctly then the first node will contribute a $+1$ to the output which is cancelled out by the -1 from the second node. Similarly if the second node successfully stores a pattern with target $+1$ then both nodes will contribute $+1$ to the output node otherwise the contributions from the two nodes cancel each other out. If the threshold at the output node is zero, patterns contributing two $+1$'s or two -1's will give the correct output. To handle those patterns which contribute zero to the output it is necessary to grow further hidden nodes. In growing further hidden nodes we must avoid disrupting patterns correctly stored by previous hidden nodes. This leads to two strategies. In the first (section 2.1) we grow a single-hidden layer. With each $\oplus$-dichotomy we must store the original P^+ pattern set and those members of P^- not stored at previous $\oplus$-dichotomies (and vice versa for $\ominus$-dichotomies). The second approach (section 2.2) involves alternating dichotomies in the hidden layer and we only store those patterns not stored correctly at previous pairs of dichotomies. Since learnt patterns (i.e. two $+1$'s or two -1's) are discarded from the training set of succeeding dichotomies we avoid disrupting previously learnt patterns by introducing a cascade structure of linear nodes or a tree of thresholding nodes between the hidden layer and output.

2. The Algorithms

2.1. Neural networks with a single-hidden layer. To generate a feed-forward neural network with a single hidden layer we proceed as follows. Let P_i^+ and P_i^- be the pattern sets at hidden node i then:
1. Perform a $\oplus$-dichotomy with the current training set (see 2.3 below). For hidden node i the training set P_i^+ is equal to the original P^+ whereas P_i^- only consists of members of P^- previously unstored at earlier hidden nodes inducing a $\oplus$-dichotomy. We repeatedly grow hidden nodes, iterating this step until all patterns belonging to P^- are stored.
2. Similarly we construct a set of hidden nodes inducing $\ominus$-dichotomies. For hidden node i the training set P_i^- is equal to the original P^- whereas P_i^+ only consists of members of P^+ previously unstored at earlier hidden nodes inducing a $\ominus$-dichotomy. We repeatedly grow hidden nodes and iterate this step until

all patterns belonging to P^+ are stored.
3. We use a threshold at the output node equal to the difference between the number of hidden nodes inducing $\oplus$-dichotomies and the number inducing $\ominus$-dichotomies.

2.2. Cascade Architectures and Tree-Structured Networks. An alternative strategy is to grow pairs of hidden nodes with alternating $\oplus$ and $\ominus$-dichotomies. Patterns learnt by a pair of hidden nodes (two $+1$'s or two -1's) are discarded from the training set of succeeding pairs of hidden nodes. We can avoid disrupting previously stored patterns by introducing a cascade structure of linear nodes between the hidden layer and output node. This architecture is illustrated in Fig. 1 (a link between two nodes indicates a connection with a corresponding weight value always fixed at $+1$; the numbers indicate the order in which hidden nodes are grown). If the first two hidden nodes output $(+1, +1)$ or $(-1, -1)$ then the first linear node feeds a $+2$ or -2 to the output and succeeding linear nodes, thereby inhibiting the latter from sending erroneous contributions to the output node. If these nodes output $(+1, -1)$ or $(-1, +1)$ then the first linear node outputs 0 and consequently the output of the network is influenced solely by the remaining hidden nodes. Instead of this cascade structure we can also use the tree structure of threshold nodes illustrated in Fig. 2 (the numbers indicate the order in which hidden nodes are grown in the initial hidden layer). The number of hidden nodes can be odd for both architectures. We can now describe the main steps in the algorithm. Starting from the original training sets P^+ and P^-:
1. At hidden nodes i and $i+1$ we perform $\oplus$ and $\ominus$-dichotomies respectively.
2. If the previous pair of hidden nodes failed to store some of the pattern set then we create training sets P^+_{i+1}, P^-_{i+1}, P^+_{i+2} and P^-_{i+2} for a further two hidden nodes inducing a $\oplus$ and $\ominus$-dichotomy respectively. These training sets consist of patterns previously unstored at earlier pairs of hidden nodes. Steps **1** and **2** are iterated until we obtain a separation (this can occur at an odd-numbered node).
3. Finally we grow the cascade or tree architectures shown in Figures 1 and 2.

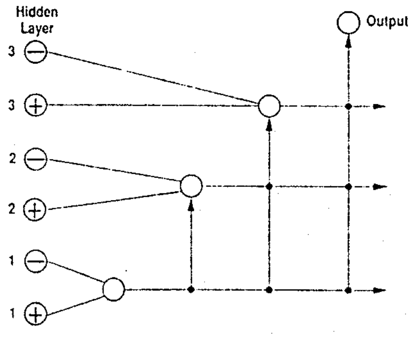

FIG. 1 CASCADE ARCHITECTURE

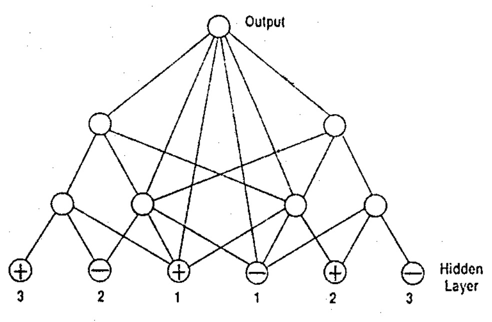

FIG. 2 TREE ARCHITECTURE

2.3. The Dichotomy Procedure. To implement these algorithms we need an efficient procedure for obtaining the dichotomies at each hidden node. To obtain a $\oplus$-dichotomy we use the following procedure:
(1) We use an algorithm to find a set weights W_{ij} between the input nodes j and hidden node i. If the pattern set is not linearly seaparable the algorithm attempts to find the best linearly separable subset storing most patterns. We will discuss this step in more detail below.
(2) Using these weights we now calculate $m_i^\mu = \sum_j W_{ij}\xi_j^\mu$ for all the patterns belonging to P_i^+ and P_i^-. Among those patterns belonging to the set P_i^- we find the pattern with the largest value of m_i^μ. Suppose this is pattern $\mu = \lambda$ then we set the threshold at i equal to m_i^λ i.e. $T_i = m_i^\lambda$.
(3) For the set of patterns belonging to P_i^+ we then find if there are any patterns such that m^μ is less than or equal to m^λ i.e. $m_i^\mu \leq m_i^\lambda$. If there are no patterns in P_i^+ with m_i^μ less than or equal to m_i^λ then we have finished training the weights and thresholds leading into hidden node i and we proceed to step (5) below. However, if there are patterns in P_i^+ satisfying this inequality then we find that pattern in P_i^+ which has the smallest value of m^μ. Let us suppose this is pattern $\mu = \nu$.
(4) Among those vectors belonging to P_i^- with $m_i^\mu \geq m_i^\nu$ we find the pattern with the largest value of m^μ and assign it the value $+1$ (i.e. this pattern moves from P_i^- to P_i^+). With the new sets P_i^+ and P_i^- we return to step (1) to find a new set of weights and thresholds.
(5) We have now obtained a $\oplus$-dichotomy. For the remaining members of P_i^- the sums $\sum_j W_{ij}\xi_j^\mu$ are less than the threshold T_i whereas for patterns belonging to P_i^+ these sums are greater than T_i. However, this dichotomy may not be the best solution and consequently we can proceed with further training to maximise the number of patterns in P_i^- which are stored correctly. To do this we record the number of P_i^- patterns which were correctly stored (and associated weights and thresholds). We then discard these correctly stored P_i^- patterns and use the unstored P_i^- and the original P_i^+ as our training set, repeating steps (1)-(4). Eventually we will exhaust the entire P_i^- set and we choose the solution which stored the largest number of P_i^- patterns as the set of weights and threshold for this hidden node. Step (5) substantially reduces the number of hidden nodes generated by the algorithm described in section 2.2 though it is less useful for the single-hidden layer algorithm in section 2.1 (at least for the problems we investigated).

To obtain a $\ominus$-dichotomy we follow a very similar procedure. In step (2) we find the pattern with the smallest value of m_i^μ (for $\mu \in P_i^+$) and set the threshold T_i equal to this value of m^μ. If the pattern sets are not linearly separable we search through the P_i^+ to find the pattern which is least well stored, switch its target-value $+1 \rightarrow -1$ and iterate the sequence until a separation of the two sets is achieved.

Some patterns can lie in the hyperplanes found in the $\oplus$ and $\ominus$-dichotomies (for example in step (2) the pattern $\mu = \lambda$ lies in the hyperplane). Since we use the convention $sign(0) = +1$ it is necessary to offset the thresholds in step

(2) of the $\oplus$-dichotomy by a positive quantity $T_i \rightarrow T_i + \delta$. For binary inputs quantised ±1 and binary weights we can set $\delta = 1$. However, for analogue inputs or real weights δ should be a very small quantity.

So far the dichotomy procedure is general and a number of different learning rules can be used in step (1). In the simulations below we used the Minover algorithm [2]. Minover is a perceptron-like rule and in the simulations we used a "pocket" version to find the best solution storing most patterns. For the binary-weight solutions mentioned below we clipped the weights to ±1.

3. Simulations

In this section we will compare generalisation performance for both real and binary weights in addition to making a comparison with other algorithms. We will consider two Boolean problems, the Mirror Symmetry problem and Shift Detection, and illustrate the improvement in generalisation performance achieved with binary-valued weights. We will then briefly outline an extension of the algorithm proposed in section 2.2 to handle analogue input data.

3.1 Mirror Symmetry Problem. For the Mirror Symmetry problem the output of the network is 1 if the input bit string is exactly symmetrical about its centre, otherwise the output is -1. To investigate generalisation ability we generated 100 training patterns such that the first half of the input bit string was randomly constructed from ±1 with both components selected with a 50% probability. We define the generalisation rate as performance on a test set (of 1000 patterns) drawn randomly from the same pattern distibution as the training set (but excluding training patterns). For randomly constructed inputs the output will be -1 with a high probability. Consequently we randomly chose the two target values with 50% probability and determined the symmetry of the input strings accordingly. For the algorithm generating a single hidden layer (section 2.1) the generalisation rate was $67.3 \pm 3.8\%$ for real weights and $77.3 \pm 5.0\%$ for binary weights using the same pattern set (the average number of hidden nodes generated was 16.8 and 32.0 respectively and we used a sample of 200 networks). For the Cascade and Tree-Structured networks (section 2.2) generalisation was $68.4 \pm 3.2\%$ for real weights and $80.4 \pm 4.8\%$ for binary weights (the average number of nodes in the initial hidden layer being 5.2 and 13.2 respectively). Thus, for this problem, generalisation is better for binary weights rather than real weights and the Cascade and Tree-Structured networks generalise better than the single-hidden layer network.

3.2 Shift Detection Problem. In the Shift Detection problem we consider a network with 20 input nodes and one output node. The first 10 input nodes are a randomly constructed pattern with components ±1 and the second set of 10 input nodes is given the same pattern set circularly shifted by one bit to the left (target=+1) or right (target=-1). In our simulations we trained the network with 100 patterns and tested generalisation performance on 1000 examples drawn from the remaining patterns (samples of 200 networks were used). For the algorithm generating a single hidden layer generalisation was

$88.9 \pm 3.7\%$ for real weights and $95.4 \pm 2.3\%$ for binary weights (the average number of hidden nodes being 12.6 and 19.2 respectively). For the Cascade and Tree-Structured networks generalisation was $86.6 \pm 4.6\%$ and $91.4 \pm 3.4\%$ respectively (with average number of nodes 7.3 and 15.8). For this problem a single hidden layer architecture appears best.

For comparison with other algorithms we may quote results from Nowlan and Hinton [3] who have also used the Shift Detection problem to study the performance of a neural network with soft-weight sharing. They used an identical 20-node input network with 10 hidden nodes. 100 training patterns were used in addition to a validation set of 1000 examples (the test set was drawn from the remaining examples). For standard Backpropagation the generalisation rate was $67.3 \pm 5.7\%$ while for Cross-Validation generalisation improved to $83.5 \pm 5.1\%$. For soft-weight sharing generalisation was $95.6 \pm 2.7\%$ (5 components) and $97.1 \pm 2.1\%$ (10 components). Though the latter statistics appear marginally better our algorithms have the advantages of generating the architecture and guarranteed convergence (gradient descent methods have the disadvantage that spurious local minima proliferate in the presense of weight-sharing). Also we did not use the validation set of 1000 examples during training.

4. Discussion

In the above discussion and simulations we have used binary input data quantised ± 1. Convergence was guarranteed because it is always possible to enforce a minimal solution storing all the patterns of one target-sign and one pattern of the opposite target sign: geometrically this is equivalent to isolating one target value on the surface of a hypersphere using a tangential hyperplane. For analogue input data the pattern vectors are of arbitrary length so this construction is not always possible. However, for analogue input data it is possible to guarrantee convergence for the algorithm in section 2.3 by noting that a pattern distribution may exclude both a $\oplus$ and $\ominus$-dichotomy but either a $\oplus$ or $\ominus$-dichotomy is always possible (because there is a pattern vector or subset of vectors of maximal length). This extension has performed well on analogue data [4] (for example, for the aspect-angle independent sonar classification dataset of Gorman and Sejnowski [5] generalisation performance was $85.0 \pm 7.2\%$ exceeding their results for Back-Propagation). Finally we note that storing noisy data can lead to overfitting consequently it is also advisable to use a pruning procedure after constructing the network: appropriate prunning strategies are outlined elsewhere [4].

[1] M. Marchand, M. Golea and P. Rujan, *Europhysics Letters* **11**(1990)487-492.
[2] W. Krauth and M. Mezard, *J. Phys.* **A20**(1987)L745-L752.
[3] S. J. Nowlan and G.E. Hinton, *Neural Computation* **4**(1992)473-493.
[4] C. Campbell and C. Perez Vicente, preprint submitted to *Neural Computation.*
[5] R.P. Gorman and T.J. Sejnowski, *Neural Networks* **1**(1988)75-89.